

ALL ABOUT REXX! Develop with REXX • Extend REXX • Debug REXX • REXX Products • and MORE!

REXX

R E P O R T

Special! All-REXX Report

REXX RULES!

You will, too...
with the hottest REXX
tips at your service

Visually Sampling
DB2/2

Mike Cowlshaw
on the Joy of REXX

Rick McGuire
on Object REXX

REXX
Buyer's Guide


SPECIAL!



REXX REPORT
is brought
to you by the
publishers of

os/2
Developer

\$9.95 U.S.
\$10.95 Canada

A Miller Freeman Publication



WATCOM™ VX•REXX™

Visualize the Power of OS/2

NEW
VERSION!



WATCOM VX•REXX is an award winning, easy to use visual development environment for creating OS/2 applications with rich graphical user interfaces. VX•REXX combines a project management facility, visual designer and an interactive source-level debugger to deliver a very approachable and highly productive visual development environment.

Design Applications Visually

Create rich graphical applications quickly and easily using the visual design environment. With the visual designer, you can graphically create CUA'91 Presentation Manager interface objects, quickly customize their properties, and easily attach REXX procedures using powerful drag-and-drop programming techniques.

Integrated Development Environment

Build, test and debug your application without leaving the development environment. Then package your application as an EXE file or PM macro for royalty-free redistribution. The power of the integrated development environment and debugger can also be used with your existing REXX applications.

Powerful Open Environment

Enjoy the simplicity of event-driven programming together with the global editing capabilities essential for professional project management. WATCOM VX•REXX is open and extensible through IBM's object-oriented System Object Model (SOM) technology. You can access all standard REXX API's including DB2/2, because VX•REXX is based on the OS/2 2.x standard system REXX.

Experts Agree... "All in all, VX•REXX stands out for ease of use, versatility, and power. And at \$199, including free tech support and free application distribution, it's also a sensational value." PC Magazine, February 8, 1994.

"(VX•REXX) applications can be multithreaded and REXX is probably the easiest language in which to learn the OS/2 thread model." Software Development, November, 1993.

"VX•REXX is a great tool; it's fun and productive." PC Techniques, Dec/Jan. 1994.

Suggested Retail: \$199*

Watcom

A Powersoft Company

Watcom International

415 Phillip Street, Waterloo, Ontario, Canada. N2L 3X2 Phone: (519) 886-3700 Fax: (519) 747-4971

*Prices and specification are subject to change without notice. Price does not include freight and taxes where applicable. Prices quoted in US dollars. Watcom, the Lightning Device, and VX•REXX are trademarks of Watcom International Corporation. Other trademarks are the properties of their respective owners. ©Copyright 1994 Watcom International Corporation.

Version 2.0 Highlights

- ▶ 26 Objects including CUA'91 Containers, Notebooks, Pop-up Menus and more
- ▶ Integration and control of existing applications through DDE, keystrokes or REXX API's
- ▶ Easy to learn event-driven programming model with complete on-line documentation
- ▶ Powerful drag-and-drop programming techniques simplify programming
- ▶ Develop professional multi-threaded, multi-windowed and drag-and-drop enabled applications
- ▶ 700 page manual packed with examples and sample programs
- ▶ Include OS/2 style help and hints in your applications
- ▶ Advanced interactive source-level debugger
- ▶ Support for the Q+E database library included
- ▶ Royalty-free run-time
- ▶ Package your applications as EXE files or PM macros
- ▶ Integrated console window support simplifies migration of existing REXX programs



for OS/2

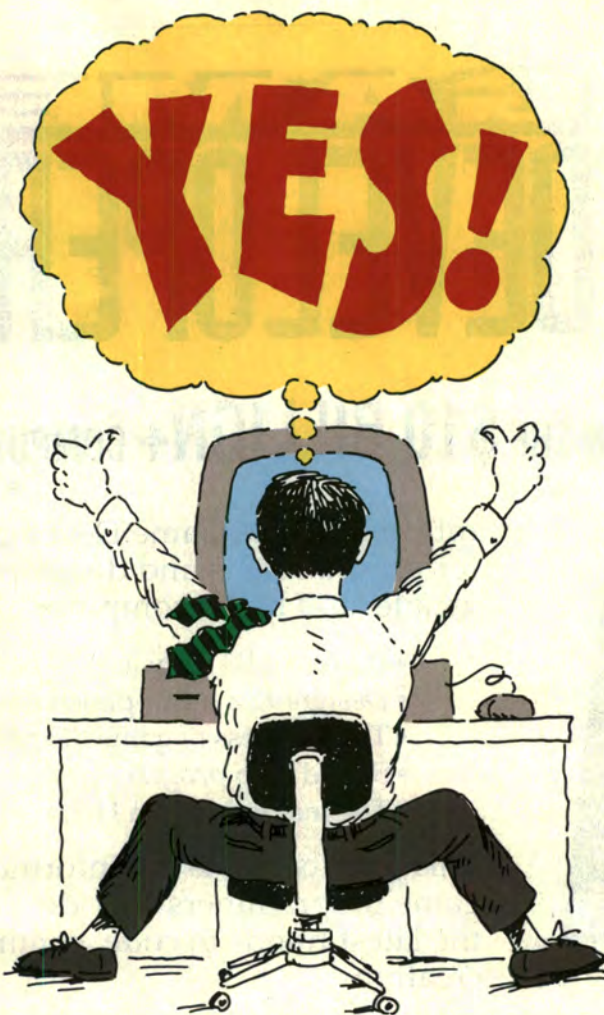


February 8, 1994
Watcom VX•REXX, Version 1.01



"The best new development tool of 1993"

1-800-265-4555



HARNESS THE POWER OF OS/2

INTRODUCING OS/2 MAGAZINE



Make the most of your PC with *OS/2 Magazine*! Every issue brings you the information you need to achieve maximum performance. From your hardware and software, to your operating system and peripherals.

OS/2 Magazine helps you harness the power of your PC. Why not flip open the next issue to timely feature articles like these:

- Common OS/2 problems and how to solve them
- How to detect OS/2 viruses—and keep your data safe
- Putting your OS/2 PC onto a local-area network
- Making multimedia applications come alive under OS/2
- Running Windows applications under the Workplace Shell

You'll also get hands-on tutorials. Expert technical advice. And unequivocally, unbiased product reviews—on HP Laserjets, S3 graphic boards and Intel fax modems, to name a few. Plus the latest news on OS/2, invaluable network support and more.

SEE FOR YOURSELF

First issue FREE! Check out the next issue of *OS/2 Magazine*. It's on us. If you like it, you can get a full year's worth (11 additional issues) for the low charter rate of \$29.95. You'll save 25% off the regular subscription rate!

Why not return the attached form today? You have so much to gain, and *absolutely* nothing to lose!

**SPECIAL
INTRODUCTORY
OFFER!**



YES! Send me a complimentary copy of *OS/2 Magazine*. If I like it, I can get a full year's worth (11 additional issues) for the low charter rate of \$29.95. I save 25% off what regular subscribers pay. If *OS/2 Magazine* is not for me, I'll simply write "cancel" on your invoice and that will be that. I owe *nothing*. I lose *nothing*. And the *FREE* issue is mine to keep.

NAME _____

ADDRESS _____

CITY/STATE/ZIP _____

Mail your order to: *OS/2 Magazine*, P.O. Box 56664, Boulder, CO 80323-6664, or call (800) 765-1291, or Fax (415) 905-2233.

Allow up to 6-8 weeks for delivery of your first issue. Canadian/Mexican subscribers please add \$10 for surface mail; overseas add \$40 for airmail. Prepayment drawn in U.S. dollars on a U.S. bank for foreign orders.

GAME DEVELOPER

TAKE YOUR TURN IN THE **\$10 BILLION+** COMPUTER GAME INDUSTRY!



Introducing . . . Game Developer, the magazine for programmers and developers of console, arcade, and home computer games.

- Profile of ID software
- Designing a multi-player game
- The business of game distribution
- Sound chip programming
- The making of Iron Helix

This publication is the information source for game programmers and developers looking for the latest trends in code, commerce and creativity.

.....**ORDER YOUR COPY TODAY!**.....

YES! Send _____ copies at \$6.95 each = \$ _____

(Foreign add \$3.00 each) \$ _____

TOTAL DUE \$ _____

Name _____

Company _____

Address _____

City/State/Zip _____

Prepayment Required

☐ Check enclosed

☐ Charge my:

☐ VISA

☐ MC

☐ AMEX

Card # _____ Exp. _____

Signature _____

Please make checks payable to Miller Freeman, Inc. Orders must be prepaid in U.S. Dollars. Please allow 3-4 weeks for delivery. Price includes domestic shipping & handling.

To order: Telephone 1-800-444-4881 -- or -- Fax 1-415-905-2233 -- or -- Mail this order form and payment to: Game Developer, P.O. Box 105448, Atlanta, GA 30348-5448

EDITOR*Dick Conklin***EXECUTIVE EDITOR***Nicole Freeman***MANAGING EDITOR***Marta Dils***EDITORIAL ASSISTANT***Diane Anderson***EDITORIAL ARTIST***Peter Altenberg***GROUP DIRECTOR***Regina Starr Ridley***PUBLISHER***Cathy Passage***ADVERTISING SALES***Angela Barnett**(415) 905-4983**Holly Meintzer**(212) 626-2275**Kristin Morgan**(212) 626-2498***MARKETING MANAGER***Susan McDonald***ART DIRECTOR/MARKETING***Christopher H. Clarke***ADVERTISING PRODUCTION****COORDINATOR***Tom Marzella***VICE PRESIDENT, PRODUCTION***Andy Mickus***VICE PRESIDENT, CIRCULATION***Jerry Okabe***CIRCULATION DIRECTOR***John Rockwell***CIRCULATION MANAGER***Lucinda Formyduval***SUBSCRIPTION INQUIRIES***U.S.: (800) WANT-OS2**Foreign: (708) 647-5960*

A SPECIAL REPORT FROM THE
PUBLISHERS OF

OS/2
Developer
THE MAGAZINE FOR ADVANCED SOFTWARE DEVELOPMENT

**REXX***REXX Motivations 4**REXX is for Lovers 8**Designing REXX-Aware Applications 33**REXX Buyer's Guide 54***DATABASE***Prototyping Database Applications with VX-REXX 18**Visually Sampling DB2/2 39***TOOLS***Debugging REXX Applications 23**Extending REXX with External Functions 47***Contact***Dick Conklin
(Editorial)**Miller Freeman
(Circulation)**Cathy Passage
(Publisher)**Marta Dils
(Managing Editor)***CompuServe***76711,1005, or
OS2DF2 Forum**71572,341**71673,1163***Internet***os2mag@vnet.ibm.com**71572.341@compuserve.com**csp@mfi.com**mdils@mfi.com**OS/2 Developer: Vol. 6, No. 5, Summer 1994*

Subscription information: U.S.: One year (six issues), \$39.95. Canada, Mexico, and international surface mail, add \$16 per year for postage. Canadian GST no. 124513185. International air mail, add \$30 per year for postage. Foreign orders must be accompanied by payments in U.S. funds. For new orders and customer service, call (800) WANT-OS2 (800-926-8672) or (708) 647-5960 (fax: 708-647-0537). IBM employees and branch office customers can subscribe to *OS/2 Developer* through IBM Mechanicsburg's Systems Library Services (SLSS) using *OS/2 Developer's* order number: G362-0001. POSTMASTER: Please send address changes to *OS/2 Developer*, P.O. Box 1079, Skokie, IL 60076-8079. Allow eight weeks for subscriptions to begin. *OS/2 Developer* (ISSN 1073-0729) is published bimonthly by Miller Freeman Inc., 600 Harrison Street, San Francisco, CA 94107, (415) 905-2200. Application to mail at second-class postage rates is pending at San Francisco, CA and additional mailing offices.

OS/2 Developer has applied for BPA membership.

© Copyright 1994 by Miller Freeman Inc. PRINTED IN THE U.S.A.



Miller Freeman, Inc.
A MEMBER OF THE UNITED NEWSPAPERS GROUP



*In this article, you'll find a brief history of the REXX programming language, how and why it was written, its main advantages and disadvantages, and a look to the future. By **MIKE COWLISHAW***

REXX Motivations



Mike Cowlshaw

REXX use is growing rapidly in the OS/2 world, with more and more applications providing REXX interfaces. With the forthcoming major upgrade in the form of Object REXX, REXX will become the scripting language of choice for object-oriented tools as well as for today's more classical applications.

A BRIEF HISTORY OF REXX

Before computers were inexpensive enough for widespread individual ownership, they usually had to be shared by many people. IBM's most successful time-sharing operating system in the 1970s and 1980s was the VM/370 (Virtual Machine/370) operating system. This operating system provided a personal computer for each of its users through the Virtual Machine concept—very similar to the virtual DOS machines in OS/2.

Any System/370 operating system can run in a VM virtual machine; the most popular for interactive development is the Conversational Monitor System (CMS). CMS is a single-user operating system that borrowed many of its features from earlier systems such as Multics, including the concept of controlling the system by commands. CMS's command language is

still one of the simplest and most readable ever devised.

Like other command-driven operating systems, CMS provides the ability to "wrap up" commands with a programming logic; initially, these simple programs were written in a language called EXEC. This language, though not much more advanced than the PC-DOS "Batch" language, allowed many enhancements and new commands to be written rapidly and much more easily than in the native assembler language used for writing low-level commands.

In the 1970s, C.J. Stephenson and others at the IBM T.J. Watson Research Center realized that, if applied consistently, this concept of a command programming language was extraordinarily powerful: a single language could provide the extension language—or macro language—for a wide variety of applications. They took the EXEC language and generalized and improved it for this enhanced role; the new language was called EXEC 2.

EXEC 2 proved the concept of a general macro language. It was used mostly for writing system commands and macros for a wide variety of editors. Its interpreter was, and probably still is, the finest example of efficient and robust



System/370 assembler code.

However, EXEC 2 (like its predecessor and most other macro languages of the 1970s and even early 1980s) assumed that macro programs would be mostly commands, with relatively little logic and variables. Accordingly, it was designed to allow commands (literal strings, usually in upper case) to be written plainly, whereas language keywords and variables were identified by a prefix of an ampersand. A command, followed by a test of its result, might look like this:

```
COPYFILE &FNAME &FTYPE &FMODE = BACKUP =  
&IF &RC GT 0 &TYPE Copy failed with return  
code &RC
```

This style, while adequate for simple commands, proved cumbersome for the large and complex programs and macros that were soon being written in EXEC 2. It became clear that a new language was needed, one based on the more classical syntax and semantics used by languages in the tradition of Algol, Pascal, and PL/I, yet including the command and string programming facilities that EXEC 2 had proved to be so effective and powerful.

This new language, initially called REX (because the name sounded nice) was very much driven by the desire to make programming easy. It borrows most of its features from other languages, especially PL/I and EXEC 2, but these features are modified or expressed in ways that make them easy to use (but not necessarily easy to implement!). The code fragment shown previously would look quite different; literal strings are quoted, but language keywords and variable names are not obfuscated by special characters:

```
'COPYFILE' fname ftype fmode '= BACKUP ='  
if rc>0 then say 'Copy failed with return code' rc
```

This difference between the two languages becomes more striking as the complexity of the program increases.

The first specification for the language is dated March 29, 1979. This was written before any implementation was even designed, and it was circulated to a number of people for comment. This began the tradition of documentation before implementation that characterized the development of REXX. This first specification included three sample programs written in REXX to show how the language would look; those programs would seem familiar to today's REXX programmers, although some details have changed.

My first implementation of REXX was made freely available over IBM's worldwide internal network in late 1979 and rapidly became popular. The network made it easy for people to exchange ideas and make suggestions for improvements. Also, because the language was limited to one (large) organization, it was possible to make some rather large changes in response to these suggestions. For example, one change, in which the ARG instruction replaced an earlier ARGS instruction, required updating hundreds of thousands of REXX programs.

As a result of this direct feedback, the language quickly evolved to meet the needs of its users through a number of releases over the next few years. By 1982, it had become essentially the language known in OS/2 2.1 today; its name gained an "X" to avoid any confusion with other products. REXX was included in the third release of IBM's VM/System Product, shipped in 1983.

It was soon discovered that IBM's customers liked the language just as much as the internal users did. Only two years later, the first non-IBM implementation (by the Mansfield



Software Group for PC-DOS) became available; this product, Personal REXX, is now sold by Quercus Systems for DOS, Windows, and OS/2.

In 1987, IBM announced that REXX was to be the procedures language for its Systems Application Architecture (SAA), which was followed by implementations for a number of operating systems, including MVS/TSO, AS/400, and (in 1989) the extended edition of OS/2 1.2.

The first REXX compiler was developed at IBM's Vienna Laboratory, following research by the IBM Haifa Scientific Center, and was delivered to customers in 1989. By 1990, there was sufficient interest in the language to justify the first international REXX Symposium for Developers and Users, organized by the Stanford Linear Accelerator Center in California. This symposium has been held annually since then.

REXX became part of the base OS/2 product with OS/2 1.3; this version was the first implementation of REXX 4.00 defined and published in 1990. Enhancements to performance and to the REXX function packages provided with OS/2 were added for OS/2 2.0. And in OS/2 2.1, the multimedia component of OS/2 provided commands that allow the control of multimedia devices from REXX.

Recent versions of REXX from IBM include REXX for the AIX/6000 operating system, REXX for NetWare, and REXX for the CICS transaction monitor. REXX is now available for most significant operating systems and from several vendors, not just IBM. Freeware or shareware versions, often excellent, are also available for PC-DOS, UNIX, and other systems. Work toward an ANSI standard for REXX has been in progress since 1991, with a publication target date of 1995.

ADVANTAGES

REXX is a programming language that was developed for its users rather than for the convenience of its implementers—those who implement its compilers and interpreters. For example, it hides the underlying mechanisms of hardware from the programmer, except where exposure is absolutely necessary and appropriate.

Further, in the first five years of its life, the core of the language was defined and implemented by a single person (based on feedback from hundreds of early users) and was constrained by an explicit design philosophy. This has led to a language that is coherent, does what its users want it to do, and has a sound base for future evolution.

This design approach gives REXX several advantages over older languages:

- REXX programs can be made very readable, since there is minimum required punctuation, special characters, or notations. For example, this is a complete, runnable REXX program:

```
/* This program, hello.cmd,  
displays a greeting */  
say "Hello World!"
```

- REXX has only one data type, the character string, so no declarations are needed. This makes writing programs in the language attractive and productive, as the programmer doesn't have to worry about underlying hardware representations. REXX operators are optimized for common string operations such as concatenation and are supported by powerful parsing and word instructions and functions.
- REXX arithmetic is defined as decimal arithmetic, with precision selected by the programmer rather than by the underlying hardware, as in:

```
/* Test some arithmetic */  
numeric digits 40  
say "One seventh is" 1/7
```

which would display:

```
One seventh is  
0.1428571428571428571428571428571429
```

Exponential notation is supported in scientific form and in the multiple-of-three form usual in engineering or financial applications. Results of arithmetic operations therefore match users' expectations far better than results from the binary arithmetic used by most other languages.

- REXX puts no inherent limits on the size of strings (including those that represent numbers). Again, this removes many of the headaches that can often plague programmers.
- REXX is a relatively small language. This makes it approachable and easy to learn. Even the Object REXX under development adds only a few new constructs to the core language.
- REXX was specifically designed to be a general-purpose scripting and extension (macro or glue) language. That is, it lets users easily tailor and enhance advanced software systems. Like most human-written languages, REXX is based on simple character strings. It also has special mechanisms for rapid environment switching and simple error handling that are especially suited to this use.
- REXX has no globally reserved words. This allows robust programs to be written that will not be invalidated by future additions to the set of language instructions. In addition to freeing programmers from having to learn keywords they do not use, this is also extraordinarily important when using the lan-



guage for application extension.

Software vendors can distribute macros written in REXX which, even though they are processed in source form, can remain almost immune to changes in the REXX language itself. This benefits the user, since installing a new level of REXX (which is usually part of the underlying operating system) is unlikely to break the macros provided with an application or written by the user. It also benefits the software vendor, since much less support and fewer updates will be needed.

- REXX is very system-independent. This gives it the advantages of portability and wide application. People need to learn fewer programming languages; there is no longer the need for a new command programming language for every application and operating system.
- Finally, REXX has a number of unusual features, such as associative arrays and dynamic variable scoping, that make many algorithms much easier to design and implement.

These advantages have led to the rapid acceptance of REXX as a language for procedure automation, application extension, and scripting. Its users cover the whole spectrum of programmers, from secretaries or workstation users who just wish to customize their environment or a spreadsheet in a flexible way, to professional programmers writing or prototyping whole subsystems of software in REXX.

DISADVANTAGES

Inevitably, designs that provide for human variability and preferences are more costly to implement than those that ignore human nature. The most obvious cost is in performance. For exam-

ple, REXX's decimal arithmetic will be slower than binary arithmetic until manufacturers start providing hardware support for user-friendly arithmetic.

REXX's dynamic nature means that, even with advanced compilers, programs in REXX are usually slower than those written in languages that move most of the burden of data conversion and typing onto the programmer.

Even so, as hardware speeds increase exponentially, REXX becomes useful for more and more applications. To illustrate the magnitude of this change, the first REXX implementation for PCs ran at about 115 instructions per second on the original IBM PC. Today, on an IBM PS/2 9595 using the Intel Pentium chip running at 66MHz, OS/2 REXX runs at over 40,000 instructions per second—enough for substantial processing at human response rates.

The second disadvantage of REXX is that it is relatively hard to implement. This is a task that is done infrequently, however, and the investment pays off rapidly in improved user programs and productivity.

Finally, REXX's data type, the string, is not well supported by many other languages, so there can be extra cost involved in data conversions when REXX invokes or uses programs written in "lower-level" languages such as C or BASIC. This cost can be important when small programs with little function are called, but if significant processing is to be done, data conversion on entry and return is relatively inexpensive.

THE FUTURE

Programming languages are among the most enduring features of computer science and the computer industry; they take a decade or more to become established,

and can then survive for generations of programmers and operating systems. REXX is now widely recognized as the leading extension and macro language.

In its original form, sometimes called Classic REXX, REXX has broken new ground in usability and productivity. The next major phase of development is Object REXX. Object REXX has been evolved over several years and promises to bring REXX's traditional ease of use to the object-oriented world. For example, in OS/2, Workplace Shell objects can be used as the target for messages sent from Object REXX as easily as calling a built-in function.

Importantly, users and programmers can gradually start to make use of messages in ordinary OS/2 CMD programs; it will no longer be necessary to embrace a whole new programming style to reap the benefits of an object-oriented system. However, for the experienced programmer, Object REXX provides a first-class object-oriented programming system with the usual concepts of inheritance, polymorphism, and encapsulation. Added to that is an exceptionally simple mechanism for concurrency—and other enhancements that will be useful for both object-oriented and classic programming.

Mike Cowlshaw, IBM Fellow, is the creator of the REXX language. He has long been interested in the human aspects of computing, working on the design and implementation of languages, editors, displays, image processing systems, and text formatters.

Today, he programs almost exclusively on OS/2, writing programs such as PMGlobe to explore interactive techniques. His current technical interests (in addition, of course, to REXX) include user interfaces, lightweight computers, and neural networks.



*This article examines a new version of REXX that is currently being tested at IBM's Glendale Programming Laboratory in Endicott, N.Y. It focuses on how the language has been altered to become object oriented while maintaining compatibility. By **RICK MCGUIRE, STEVE PRITKO, GARY COLE, MARK CRAMER, and MICHAEL BARYLA***

REXX is for Lovers

If you like programming, you will love REXX. REXX has the kinds of features that can turn a romantic interlude with a programming language into a lifelong relationship: a tiny but very expressive syntax; an unbelievably simple data model, "everything is a string"; an operating system interface that is so unobtrusive it is invisible; and interfaces complete enough to build complete programming environments without changes to the language or to its implementation.

You might also be having an affair with SmallTalk, but for different reasons. SmallTalk seduces you with its object orientation. You can write programs that model real objects in a much more direct way than you can with any procedural language, even REXX. "If only REXX was object oriented," you confide, "I'd be ready for a commitment."

A new version of REXX, tentatively dubbed Object REXX, ends the love triangle. This new REXX is as beautiful as the old REXX—REXX is extended, not changed. All the existing programming interfaces are implemented so the new REXX runs all existing REXX programs, uses all existing function packages (REXXUTIL, MCI API, and so on), and works with all existing REXX program-

ming environments (VXREXX, VisPro, and GPFREXX).

Some new features include language extensions that support object-oriented programming; intuitive concurrency support based on objects; and clean integration with OS/2's System Object Model (SOM), which lets REXX transparently access objects written by other languages.

We will examine some of these new features, starting with the current REXX and how it was changed to become object oriented. Object REXX, though not yet available, is currently being tested at IBM's Glendale Programming Laboratory in Endicott, N.Y., and at selected customer sites.

THE OBJECT REXX MAKEOVER

How was it possible to transform REXX into an object-oriented language while retaining compatibility? The trick was in the handling of variables. Currently, all REXX data are typeless strings. These strings represent character data and can represent numeric data as well. A REXX variable is an undeclared pointer to a string. For example, the instruction:

```
name = 'Fred'
```

creates a string with the value 'Fred' and



gives the variable `name` a reference or pointer to the string. You do not need to allocate or release storage for the string; REXX does it for you. If you assign a new string to the variable, REXX automatically reclaims the storage for the old string in a process called garbage collection.

In the parlance of object-oriented languages, you might say that the current REXX is a language with one object class (namely, strings). Individual strings, such as `'Fred'`, would be considered instances of the string object class.

An examination of the object-oriented SmallTalk language reveals many similarities in variable handling. SmallTalk variables are also undeclared and are pointers. But instead of pointing to character strings, the variables point to SmallTalk object instances. The SmallTalk instruction:

```
name := 'Fred'.
```

creates a character string object and assigns the variable `name` to point to the object. Like REXX, SmallTalk performs garbage collection, reclaiming storage for objects that are no longer referenced. Unlike REXX, however, SmallTalk lets you create classes of objects beyond the character string. For example, the instruction:

```
a := Array new: 4.
```

creates an array object and assigns it to the variable `a`. The token `Array` is a SmallTalk class object that builds new array instances. The token `new:` tells the `Array` class object to create an array with four elements.

The same capability was added to REXX. The first step in making REXX object oriented was letting REXX variables reference classes of objects other than strings. The REXX string becomes

merely one of many classes that you can create.

The next step was redefining the entire REXX language as a series of object interactions. The redefinition does not cause any incompatibilities with the old, or classic, REXX. For example, the REXX instruction:

```
name = 'Fred'
```

has exactly the same effect as before. All REXX functions and instructions in which `name` is used will work the same as they did in classic REXX. A `SAY` instruction, for instance, would still display the string `'Fred'`.

The final step in making REXX object oriented was the introduction of a new operator for object interaction. In the previous SmallTalk example, the space between the tokens `Array` and `new` is a message send operator. The message send operator invokes a capability or method of the `Array` class object. REXX already uses a blank space for the concatenation operator, so a different syntax was needed for a message send. We chose the `~` (tilde) character, which we frequently refer to as the twiddle. The REXX equivalent to the the SmallTalk message is:

```
.array~new(4)
```

The token `.array` is a reference to the REXX array collection object. The twiddle sends the message `new` to the array collection object. An argument of 4 is also sent. The `new` method for the array class object is invoked and constructs a new four-element array.

By using the twiddle character and new REXX classes, every REXX function can be redefined in a similar way. For example:

```
x = substr(name, 2, 3)
```




is redefined as:

```
x = name~substr(2, 3)
```

That is, the object pointed to by name (the string 'Fred') is directed to extract a substring of itself, starting at the second character, for three characters. The substr method returns a new string object as a result (the string red). Similarly, all the REXX operators are really message sends. An arithmetic operation such as:

```
x = 1 + 2
```

can be reworked into the message send syntax. The message send equivalent is the admittedly not very usable:

```
x = 1~'+'(2)
```

This operation sends the + (plus) message to the object 1, passing the argument 2. The object 1 processes the addition operation. We would never suggest that you use this syntax in normal REXX programming, but this "under the covers" message definition provides some very powerful extensibility to the REXX language.

CLASS ACTS

So far, we've dealt with only REXX strings with a brief allusion to a REXX array class. Let's now extend REXX a little by making a new object class we can use within our REXX code. For our example, we will create a complex number class.

Complex numbers are expressed as a combination of a real number and an imaginary number that is multiplied by the square root -1, which is named i. The imaginary number with a real part of 5 and an imaginary part of 2 is written with the notation 5+2i. There are formula that define all the standard arithmetic operations on complex

numbers. Let's model complex numbers with a REXX class.

To define a new REXX class, we use a new type of REXX instruction called a directive. The directive instruction for defining a complex number class is:

```
::class complex      /* create a complex
                        number class */
```

This instruction creates a new REXX class called complex. To create a new instance of the complex class, we send a new message to the complex class object:

```
x = .complex~new(5, 2)
```

This creates a new instance of a complex number object and invokes a method or routine named INIT, passing any arguments on the new message. We need to create an INIT method to finish creation of the object, which is illustrated in Figure 1.

The INIT method uses some new REXX instructions. The ::method directive creates an object method named INIT that is part of the complex number class. Every complex number we create calls the INIT method to initialize the object state.

The second instruction, expose real imaginary, accesses the state information of this object instance. The variables real and imaginary are object variables. The values assigned to these variables will exist for as long as this complex object exists (until it is reclaimed by garbage collection). Every complex number that we create has object variables named real and imaginary, but each individual complex number has its own unique values assigned to these variables.

The use arg first, second instruction accesses the argument expressions passed to the method. The instruction creates a one-to-one mapping of method variable to a method argument. Using our example, first is assigned the value 5, and second is assigned the value 2. The variables first and second are local method variables. Any values assigned to these variables will go away when the INIT method ends. The variables first and second are not actually needed in this method. The use arg instruction can also assign values directly to object variables:

```
use arg real, imaginary
```

If we try to display the com-

```
::class complex      /* create a complex number class */
::method init        /* initialize a complex number */
expose real imaginary /* access the state variables */
use arg first, second /* get the arguments */
real = first          /* assign the "real" part */
imaginary = second    /* assign the "imaginary" part */
```

Figure 1. INIT method

```
::method string      /* construct the object string value */
expose real imaginary /* access the state variables */
return real~'+~'imaginary~'i' /* format as "n+mi" */
```

Figure 2. Addition of STRING method



plex number with the SAY instruction say x, we see the string a complex. The complex number itself is not displayed because the SAY instruction invokes the STRING method, which returns the string value of an object.

Every REXX object has a string value—the default object string value is the name of the object class. To display the complex number as 5+2i, we'll need to add our own STRING method to the complex class, as shown in Figure 2.

The ::method directive creates a STRING method for the complex class. The expose instruction in this method accesses the same object variables created in the INIT method. Any method defined for the COMPLEX class can access the object variables with the expose instruction. The instruction say x will now display the string 5+2i.

Now let's add arithmetic to the

complex class. As previously shown, the instruction:

```
a = x+y
```

can be defined as:

```
a = x~'+'(y)
```

To add addition capability to complex numbers, we need to create a '+' method. The directive instruction ::method '+' will create the addition method, illustrated in Figure 3.

As before, the expose instruction accesses the object variables real and imaginary, and use arg accesses the argument passed to the '+' message. Complex number addition adds the real parts of the two numbers and the imaginary parts of the two numbers. The expose instruction gives us the two parts of the target number (x), but

we have a problem with the second number (y, which is the argument). The values in the object variables of one instance (y) cannot be directly accessed by another instance (x). In other words, the real and imaginary values associated with instance y cannot be directly accessed by instance x. This isolation of object variables is known as encapsulation.

We could access the values of the argument object by sending the message STRING to the object adder and use PARSE to split it apart again, but there is a better way. By adding methods that allow the real and imaginary parts to be retrieved, we can easily retrieve these values from another method, which is shown in Figure 4.

These methods did not expose all the object variables. A method needs to expose only the variables it uses. The object state data consists of all values assigned to exposed variables in all methods of the object.

Now that we have access to the state data of the other object, we can complete our addition method (Figure 5).

The methods -, *, and / are added in similar fashion, creating an object that is polymorphic with the string class for the arithmetic operations. The sequence:

```
x = .complex~new(5,2)
y = .complex~new(3,4)
say x+y
```

will display the value 8+6i. The

```
::method '+'                /* complex number addition */
expose real imaginary      /* access the number parts */
use arg adder              /* get the second addition number */
```

Figure 3. Creating the addition method

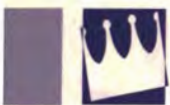
```
::method real                /* retrieve a complex number real part */
expose real                  /* expose the real value */
return real                  /* return the real part */

::method imaginary           /* retrieve a complex number imaginary part */
expose imaginary             /* expose the imaginary value */
return imaginary             /* return the imaginary part */
```

Figure 4. Methods to retrieve real and imaginary parts

```
::method '+'                /* complex number addition */
expose real imaginary      /* access the number parts */
use arg adder              /* get the second addition number */
tempreal = real + adder~real /* add the real parts */
tempimaginary = imaginary + adder~imaginary /* add the imaginary parts */
return .complex~new(tempreal, tempimaginary) /* create a new complex */
```

Figure 5. Completion of addition method



complex numbers can now be used in normal arithmetic.

SUBCLASSES AND INHERITANCE

Now let's build on what we've created. In mathematics, vector arithmetic is quite similar to complex number arithmetic. A vector is represented by a pair of numbers, and the equations for addition, subtraction, multiplication, and division are the same as the equations for complex numbers. The major difference is the notation used to represent a vector value. We would like a vector object for the values 5 and 2 to display as (5,2).

One approach to creating the vector class is to copy all of the complex class methods to a new class called vector. However, we would need to maintain two sets of very similar code, and we would need to apply any enhancements (or even bug fixes) to both sets of code. Object-oriented programming provides a better way of managing this. When we create our vector class, we use a slightly different `::class` directive:

```
::class vector subclass complex
/* create a vector class */
```

The `subclass` keyword lets the vector class inherit the code used for the complex class. The inheritance gives vector all of the methods we defined above. We're not even going to define an `INIT` method for vector, but rather will let the complex `INIT` method handle the setup. For the complex class, the `STRING` method formats the object into the form `5+2i`. As we are currently defined, the vector class will display the same way. We can change this by creating a new `STRING` method for the vector class. The new vector class now becomes

```
::class vector subclass complex
/* create a vector class */
::method string
```

```
/* replace the string method */
return '('self~real', 'self~imaginary')'
/* reformat the vector value */
```

`self` is a special variable that holds the value of the object for which the method is running. Sending the messages `real` and `imaginary` to `self` returns the values for the object that the method is currently processing. But why did we use the messages `real` and `imaginary` instead of exposing the variables directly? We needed to use the messages because the vector class has its own set of object variables (called a scope) that is separated from any classes from which it inherits. The variables `'real'` and `'imaginary'` refer to different values when used in a vector method or a complex method. Once again, the object state data is encapsulated. Vector addition now looks like this:

```
x = .vector~new(5,2)
y = .vector~new(3,4)
say x+y
```

and it will display the value (8,6). Our vector class has all the methods defined for complex but has replaced the `STRING` method to format the values differently. We can also add to the vector class new methods, such as `LENGTH`, that apply only to vector objects.

THE REXX OBJECT MODEL

The complex and vector classes are factories for producing complex and vector objects, called instances. The directives used in the definitions of the classes capture three key relationships. First, a complex object "can perform" addition. "Can perform" relationships are expressed by `::method` directives. Second, a complex object "has a" real part and an imaginary part. "Has a" relationships are expressed by the `expose` statements in `::method` directives.

Finally, a vector "is a" kind of complex object. "Is a" relationships are expressed by inheritance, in this case using the subclass keyword in a `::class` directive.

In the example, vector is a subclass of complex. We also say that complex is a superclass of vector. Classes inherit both methods and object variables from their superclass. In the example, a vector "is a" complex, so it "has a" real and imaginary part and "can perform" addition. Classes normally specialize their superclass by adding new methods, overriding existing methods, or adding new object variables. Inheritance captures the similarity between objects, requiring us to define only the differences.

Although the `::class` directive for the complex class did not specify a superclass, complex does inherit from the superclass of all REXX classes: `.object`. In fact, all classes inherit either directly or indirectly from `.object`. This allows a very useful common set of behaviors to be included in all instances, including very basic methods, like `STRING`, as well as more powerful ones, like `START`.

We call any set of inheritance related classes a class hierarchy. The hierarchy in the example is:

```
object
  complex
    vector
```

where indentation indicates subclassing. Many classes are included in the basic REXX class hierarchy, forming a library of reusable code:

```
object class
  stream class
  fileys class
  set class
  bag class
  table class
  supplier class
  string class
  queue class
```



```
message class
list class
integer class
directory class
array class
class class
```

Although class hierarchies are very powerful, sometimes it is useful to add some generic behavior to a target group of classes. Normally, we would do this by adding methods to the target classes themselves or to a superclass common to the target classes. Sometimes, we'd like to just pick and choose a few classes to receive the new behavior, rather than whole branches of the class hierarchy. Classes designed to supply this sort of generic behavior are called *mixins*.

Usually, classes are intended to be factories to create many instances. Sometimes, however, we need to create a single one-of-a-kind object. Creating a class to create a single instance seems to be more work than necessary. To save programmers this extra work, REXX supports instance-based inheritance, as shown in the following example:

```
oneoff = .object~new
oneoff~setmethod('STRING', 'return "I
    am unique!"')
say oneoff
```

ACCESSING THE SYSTEM OBJECT MODEL

Object REXX lets you access any objects or classes that have been created using OS/2's System Object Model (SOM). To use a SOM class in your Object REXX program, you must first import the SOM class.

Once the SOM class is imported, you can use it just as you would use any other Object REXX class.

Let's assume we have available to us *Animal* and *Dog* SOM

classes, where *Dog* is a subclass of *Animal*. The *Animal* class introduces two new methods *SPECIES* and *INFORMATION*. The *Dog* class overrides both and introduces one new method *SPEAK*.

There are two ways to import a SOM class. The first is through

the use of directives, as follows:

```
::class Dog external 'SOM Dog'
```

The `::class` directive imports the *Dog* class. The name given to the Object REXX class is defined to be same as the name of the SOM



Supercharge Your REXX!

GAMMATECH™

REXX SuperSet / 2

Supercharge your REXX with the most complete set of REXX external functions available - GammaTech REXX SuperSet/2.

With over 300 functions from seven DLLs, the GammaTech REXX SuperSet/2 will initiate File and System operations, issue Network commands, execute Video I/O, manipulate Processes and Semaphores, regulate the MacroSpace and perform Math calculations.

Call us today at (405) 947-8080 and you'll see what REXX can do with some extra power!

- Perform EXECIO functions
- Manage LAN Server users by group or domain
- Perform LAN Server permission functions
- Manipulate LAN Server files, sessions and shares
- Interface with NetBIOS, TCP/IP and Communications Manager/2
- Open, close, query and create semaphores
- Start a thread
- Destroy a process or thread
- Copy, move or rename individual or mass files
- Dump variable pools or MacroSpace to a file
- Load, drop or reorganize functions in MacroSpace
- Perform logarithmic and trigonometric calculations
- Query hardware components
- List current system environment variables
- Write character and attribute strings
- Obtain and set cursor type and screen mode



SoftTouch Systems, Inc.
Workstation Division

1300 S Meridian, Suite 600, Oklahoma City, OK 73108
(405) 947-8080 • Fax (405) 632-6537

©GammaTech, Inc. 1991-94 All Rights Reserved

class. You can use a different name if you wish. The keyword **EXTERNAL** signifies that this is an imported class. The literal string indicates where we are to import the class from and the name of that class. In this case, we are importing the class from **SOM**, and will import the class **Dog**.

To import all of the **Dog** class, Object REXX needs access to all **Dog**'s parent classes. When Object REXX imports **Dog**, it notices that **Dog** is a subclass of **Animal** and checks to see if **Animal** has already been imported. If it has, Object REXX imports the **Dog** class. If **Animal** has not been imported, Object REXX automatically imports the **Animal** class and then imports the **Dog** class.

The second way to import a class is by sending a message to a special REXX object named **.SOM**. The **.SOM** object has an **IMPORT** method for importing classes:

```
somObjectCls = .SOM~import("Dog")
```

The argument **"Dog"** lets the **.SOM** object know the class name it is to import from **SOM**. We show the class name as a literal string, but it can also be an expression. The Object REXX class object representing this **SOM** class is returned and assigned to **somObjectCls**. The class name in Object REXX will be the same as the name of the **SOM** class.

After the **SOM** class is imported, you can use it as you would use any other Object REXX class. The following shows how to create an instance of this class (using **somObjectCls** from the second example):

```
somObj = somObjectCls~new
```

This would create a new instance of **Dog** and set that instance to **somObj**. Since the **Dog** class is defined in **SOM**, you may be won-

dering how we can be sure a **NEW** method exists. When Object REXX imports a class, it creates the Object REXX class with all the methods created for the **SOM** class as well as the methods from the Object REXX class class.

The imported class can then function as an Object REXX class as well the **SOM** class it represents. The **NEW** method is part of Object REXX's class class.

If a **NEW** method already exists for the **SOM** object, the Object REXX **NEW** method overrides it. As part of its processing, the **NEW** method invokes the **somNew** method on the **SOM** class to create the **SOM** object. **NEW** returns this newly

created object. You can use this instance of **Dog** just as you would use any other Object REXX object. You can also send it any of the **SOM** messages appropriate for the object. This statement:

```
somObj~somGetClassName
```

would return the name of the class to which the instance of the **SOM** object belongs. Likewise, we could send this message

```
somObj~somIsA(somObjCls)
```

to see if the **SOM** object is an instance or subclass of the **SOM** class **somObjCls**. We don't need to

```
::class Dog external "SOM Dog"
::class BigDog subclass Dog
::method information
  expose name                /* make the name of the dog available */
  self~information:super      /* Let my superclass (Dog) display */
                              /* the its information first. */
  say "name: "name            /* Now add BigDog specific info */
::method speak
  say 'BARK BARK BARK '
  say 'BARK BARK BARK '
  say 'BARK BARK BARK '
::method 'NAME='
  expose name
  use arg name                /* Assign the new name of the dog */
::method name
  expose name
  return name                 /* Return the name of the dog */
```

Figure 6. Adding behavior to a subclass

```
sadie = .bigdog~new           /* Create a new instance of BigDog */
sadie~name = 'Sadie'          /* Set the name of the dog to Sadie */
sadie~speak                   /* Displays: BARK BARK BARK */
                              /* BARK BARK BARK */
                              /* BARK BARK BARK */
sadie~information             /* displays: whatever Animal and Dog */
                              /* may display, as well as the name */
                              /* of this dog, 'Sadie' */
say sadie~name                /* Displays: Sadie */
say sadie~species             /* Displays: Dog */
```

Figure 7. Introducing **NAME** and **NAME=** methods



worry about any conversions from Object REXX objects (somObjCls) to the real SOM object (the Dog class itself) this is all taken care of by Object REXX.

Let's return to the Dog class we imported earlier. We've said that we can do anything with this class that we can with any other Object REXX class. So in addition to creating instances of this class, we can also make a subclass for the class and add additional behavior to it, as shown in Figure 6.

Here, we create a new class BigDog, which is a subclass of the SOM class Dog. Our SPEAK method overrides the SPEAK method for the SOM class Dog. Two new methods are added: NAME= and NAME.

Now, let's create an instance of this new class and send it some messages. Some of the methods (INFORMATION and SPECIES) are defined outside Object REXX and are invoked through SOM. Some of the SOM methods (NEW and SPEAK) are overridden by our Object REXX

class. The remaining two methods (NAME and NAME=) were introduced by the BigDog class in Figure 7.

CONCURRENCY

Webster's unabridged dictionary defines concurrent as "a happening together in time or place." For programs, concurrency is the ability to perform nondependent tasks simultaneously. Object REXX's concurrency has two forms: Inter-Object and Intra-Object.

Inter-Object concurrency is having multiple objects actively running a method and communicating simultaneously. Object REXX supports any number of objects being active at a time.

Intra-Object concurrency is having multiple methods actively running on the same object. Object REXX provides mechanisms to safely achieve asynchronous activation of methods on the same object.

To understand Intra-Object concurrency, you must understand

what is meant by an object's scope. The scope of any new object in a class is all the methods defined by the class and all the variables defined by these methods. Methods that are added to an instance of that class and the variables of these added methods define another scope.

Each scope has its own object variable pool. By default, a method has exclusive use of its object variable pool. If an object method is active, Object REXX delays running any other method needing the same object variable pool until the first method is finished. This default behavior prevents conflicts from compromising the object integrity. The single exception to sequential processing is when an object method sends a message to another method of the same object. The default concurrency would not allow the method to begin, and a deadlock would be detected. In this case, Object REXX processes the message immediately and treats it as an internal subroutine call (that is, shares the object variable pool, as illustrated in Figure 8).

Object REXX also provides deadlock detection. If the default concurrency is being used, methods activated for an object that are at the same scope level will be detected as a deadlock, and a deadlock message will be issued.

To activate methods concurrently, use the REPLY instruction or a message object. Use the REPLY instruction when you want the caller of a method to continue processing before the called method completes. In this case, both the caller and the called method run simultaneously. A result can be returned with the REPLY instruction, but results from the activated method after that point will not be returned, as shown in Figure 9.

A message object can be obtained by creating a new instance of the message class or with the

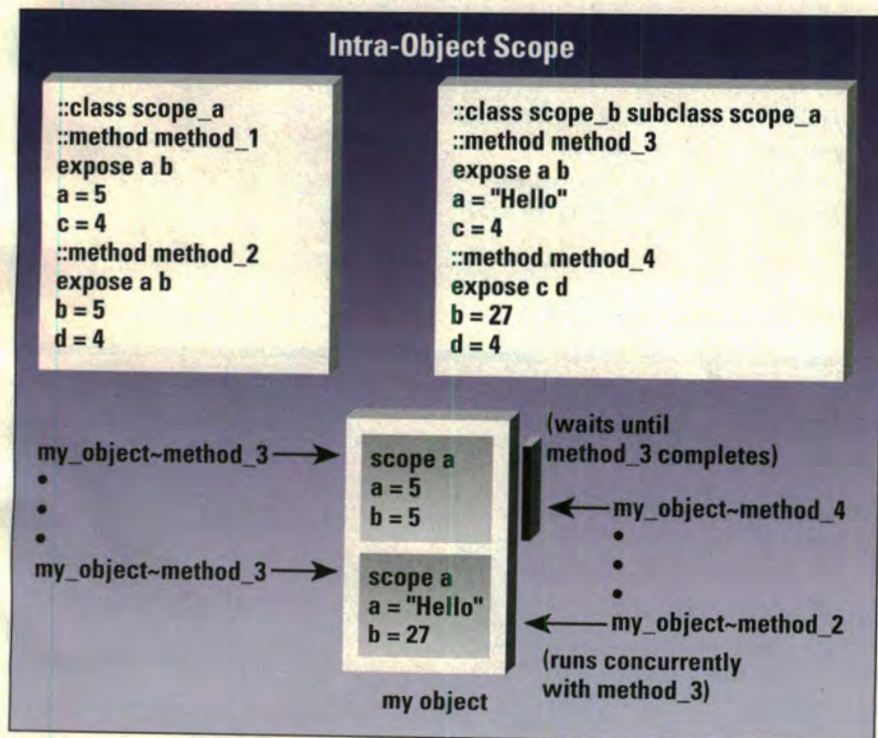


Figure 8. Example of Intra-Object concurrency

START method of the object class. The methods associated with a message object allow synchronous or asynchronous activation of a method; allow notification about the completion of the activated method; and give access to the result of the completed activated method, as shown in Figure 10.

When concurrency is needed for Intra-Object methods, the following syntax can be used:

```
--NOWAIT option of the ::METHOD directive
--SETNOWAIT method of the method class
--NOWAIT and WAIT built in routines
--GUARD instruction
```

When this syntax is used, Intra-Object concurrency of the method's state and data is up to the programmer.

The NOWAIT option of the ::METHOD directive allows a method to become active unconditionally on this object. The NOWAIT option would be coded on a ::METHOD directive:

```
::METHOD methodname NOWAIT
```

If not all the code in the method has a need to be concurrent, the CALL NOWAIT and CALL WAIT functions can be used to selectively give up or regain exclusive access to the object variable pool. CALL NOWAIT allows another method needing exclusive use of the same object variables to become active in the same object. Remember that even though two methods may be using different variables, if the methods are at the same scope, the variables are still in the same object variable pool. So if

class A method A wanted to get some information from class A method B, in order for method B to access the class A variable pool, method A would need to do a:

```
call nowait
```

prior to sending a message to method B. To regain exclusive use of the class A variable pool, method A would issue:

```
call wait
```

The GUARD instruction is used for control over specific portions of code when Intra-Object concurrency is overridden or the default Inter-Object concurrency is being used. The GUARD instruction expression is evaluated, and if it is 1

```
say_it = .sayer~new          /* get a sayer object */
say 'invoking reply_say method' /* say it's being invoked */
say_it~reply_say             /* send the message */
say 'control returned to caller' /* say control was returned */
exit

::class sayer
::method reply_say           /* reply say method */
say 'reply_say is running'   /* say when control is received here */
reply                        /* let the caller have control back */
call SysSleep 1              /* sleep so caller can run */
say 'reply_say is running again' /* say when control returned here */
return                       /* that's all we're trying to show */
```

Figure 9. An example of the reply instruction returning control to the caller

```
say_it = .sayer~new          /* get a sayer object */
say 'invoking start_say method' /* give a start indicator */
say_result = say_it~start~start_say /* start message asynchronously */
say 'caller has control'      /* signal completion */
say say_result~result         /* wait for start_say to finish */
exit

::class sayer
::method start_say           /* start_say method */
say 'start say was sent a message' /* the start say method has control */
call SysSleep 1              /* let the controller run now */
return 'start say has run'    /* return a completion message */
```

Figure 10. An example of the START method giving asynchronous control to another method



(true) the method keeps running, if the evaluation is 0 (false) the method waits until the variable(s) in the expression changes and evaluates to 1 (true). If the expression on the **GUARD** instruction cannot evaluate to 1 (true), the method never regains control (Figure 11).

We have taken a quick look at some of the new REXX extensions for object-oriented programming. These new extensions let you exploit the most powerful functions of OS/2 without having to write hundreds of lines of code. With Object REXX, you can write object-oriented programs, concurrent programs, and programs that use SOM objects. And you'll be able to write these programs with the immediacy that REXX provides.

While you are learning the new concepts, you can continue to write REXX programs as you always have. You do not need to convert to object-oriented programming overnight. In fact, Object REXX is perfectly happy if you mix traditional statements with the new object-oriented ones.

If you love REXX and have been flirting with SmallTalk, get ready to make a long-term commitment. Object REXX really will sweep you off your feet!

Rick McGuire is a senior programmer in the IBM REXX Development organization. He joined IBM in 1981 and began working as a developer on REXX for the VM/SP mainframe operating system. Since 1988, he has been concentrating on the development of REXX for all IBM systems.

Steve Pritko is a staff programmer in the IBM Object REXX Development organization. He joined IBM in 1987 and began working as a developer for Object REXX in 1992.

Gary Cole has been with IBM's Glendale lab since 1987 and is a designer for Object REXX.

Mark Cramer is a programmer in the IBM REXX development organization. He joined IBM in 1973 and has been focusing on REXX since 1992.

Michael Baryl is an advisory information developer for IBM's Glendale Programming Laboratory. He joined IBM in 1978 and has written documentation for many IBM products, including SQL/DS, VM, LANRES/400, and LAN File Services/ESA.

```
say_it = .sayer~new
say_it~second_say

say_it~first_say

say 'The guard variable will now be updated'
sayit~guard = 'first'
exit

::class sayer
::method first_say nowait
expose guard_variable
reply
self~guard_me('first')
say 'first_say is running'
guard_variable = 'first'
return

::method second_say nowait
reply
self~guard_me('second')
say 'second_say is running'
return

::method 'GUARD=' nowait
::method guard_me nowait
expose guard_variable
use arg guard_arg
guard guard_variable ^= guard_arg
return

/* get a sayer object */
/* the second say method will be the */
/* second to have output */
/* the first say method will be the */

/* first to have output */
/* let the methods run now */

/* that's all we're trying to show */

/* first say method */

/* let the caller have control back */
/* synchronize with the other methods*/
/* say when control returned here */
/* let the second method run */
/* that's all we're trying to show */

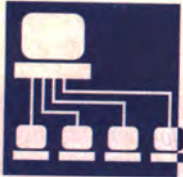
/* second say method */

/* let the caller have control back */
/* synchronize with the other methods*/
/* say when control returned here */
/* that's all we're trying to show */

/* update the guard sync state */
/* synchronize guard access */

/* wait until unlocked */
```

Figure 11. Example of a **GUARD** instruction



Database

This article provides insight into how a visual REXX tool can be used with OS/2 and DB2/2 to create database applications quickly. Journey through the creation of the Games Museum database prototype, from database structure, to using SQL from VX-REXX, to storing images in an application. BY GRACE WELCH

Prototyping Database Applications with VX-REXX



Grace Welch

When the Museum and Archive of Games at the University of Waterloo, Canada, considered downsizing its inventory database from the mainframe to a PC, it looked to two neighbors: IBM with its DB2/2 relational database system that was developed in nearby Toronto; and WATCOM, with its new VX-REXX visual development environment for OS/2 2.x that was created right in Waterloo. The two were good candidates for providing the solid, visual-development environment and powerful database platform the Museum needed. But rather than jumping into the development, the University decided to prototype the application with VX-REXX.

SYSTEM HISTORY OF THE MUSEUM OF GAMES

The Museum and Archive of Games manages the world's largest collection of games. The collection houses 5,000 pieces, ranging from ancient card games to modern pinball. Twenty years ago, the museum boasted one of the most technologically advanced collection-management systems in the industry. Each item in the museum was described in detail and recorded in a then-state-of-the-art VM/CMS-based hierarchical database called SPIRES. The conversion of the old, paper-bound ledger management system to an on-line, mainframe database greatly benefited the museum's curators. For a long time, the museum had wrestled with the problem of keeping information on each collection piece up-to-date. The museum operators needed to describe each piece as it was received, periodically adding information, such as a fading color problem and the resulting treat-

ment, to the description. The on-line database handled the accumulation of such information more conveniently than the paper system.

IMPROVING ON THE MAINFRAME SYSTEM

Two major factors caused the museum to seek a new, database solution. First, the nature of the museum's business changed. The museum shifted its focus from administration (or recording) to community and industry service. As a result, the database needed to change from an internal, administrative tool to a publicly available system. In particular, the museum wanted the database to provide easy-to-use, on-line information that included pictures of the exhibits. This way, customers could find information about games and their trends and history in a visual and interactive manner. It was not possible to achieve this ease of use within the bounds of the existing mainframe system.

Second, many university departments were downsizing their systems from the mainframe. During his search for an alternative platform, Elliott Avedon, a conservator of the museum, found a PC technology capable of meeting the museum's new requirements. Avedon viewed OS/2 and DB2/2 as powerful, user-friendly, and flexible building blocks for the Museum's database improvement. He started a project that examined the feasibility of moving the SPIRES database system to OS/2 and DB2/2.

MEETING THE CHALLENGE

Jennifer Uttley, senior researcher and developer of the University of Waterloo's Computer Systems



Group, took an interest in the museum database project. The project dealt with a problem in the area of her research: how to handle information about things, people, and events in a highly descriptive and usable manner. Within a couple of weeks, Uttley had created a working database prototype for the museum.

THE NEW LOOK OF THE GAMES MUSEUM DATABASE

The prototype database looked nothing like the SPIRES original. Instead of a text-based interface, the new database had a graphical user interface that included easily manipulated menus, drop-down list boxes, radio buttons, and push buttons. Figure 1 depicts the graphical user interface of the database prototype. The old data-retrieval method of entering arcane SPIRES commands disappeared. Now, Museum customers could retrieve data by simply pointing and clicking on the new OS/2 interface. In addition, the database prototype offered new features, including picture storage, query by example, and ranked text searches.

THE GAMES MUSEUM DATABASE STRUCTURE

The Games Museum Database prototype is based on IBM DB2/2, a relational database that groups information into tables. These tables are related to each other through primary keys.

The main table of the prototype consists of three types of fields. The fixed information fields include fields for the accession number (a unique identification number for each piece in the collection), the title of the item, the date that the item was acquired, and the market value of the item. The main table also includes a description field large enough to hold several paragraphs of information. Many smaller fields in the SPIRES database were collapsed to form this field. The new, all-inclusive description field offers an easier way to enter information that is not easily categorized. To ensure that the specific information can be easily found within the field, the prototype application includes the ability to retrieve records using a text search on the field. The main table also includes fields that contain the names of the images associated with each museum piece.

Other tables connect directly or indirectly with

Museum Database

File Search SlideShow Image

Item Name: SENET

Accession Number: 978.007.01.24

Date Acquired: 1978

Classification: OBJECT

Location: CABINET 6 - SHELF D

Source: UNKNOWN SOURCE

Image:

Description:

Alternative name for Passing Through the Netherworld: SENET Game of 30 Squares. CONTAINER: Lid shows Egyptian tomb painting of 2 people playing the game. Yellow background with blue - green - white - orange borders of Egyptian design. Plain white inside. White bottom. BOARD: divided into 30 squares each numbered clockwise direction of play. Large Egyptian design.

Search Keyword Search Clear

<< >> Add to SlideShow Exit

Figure 1. A view of the new Games Museum database application

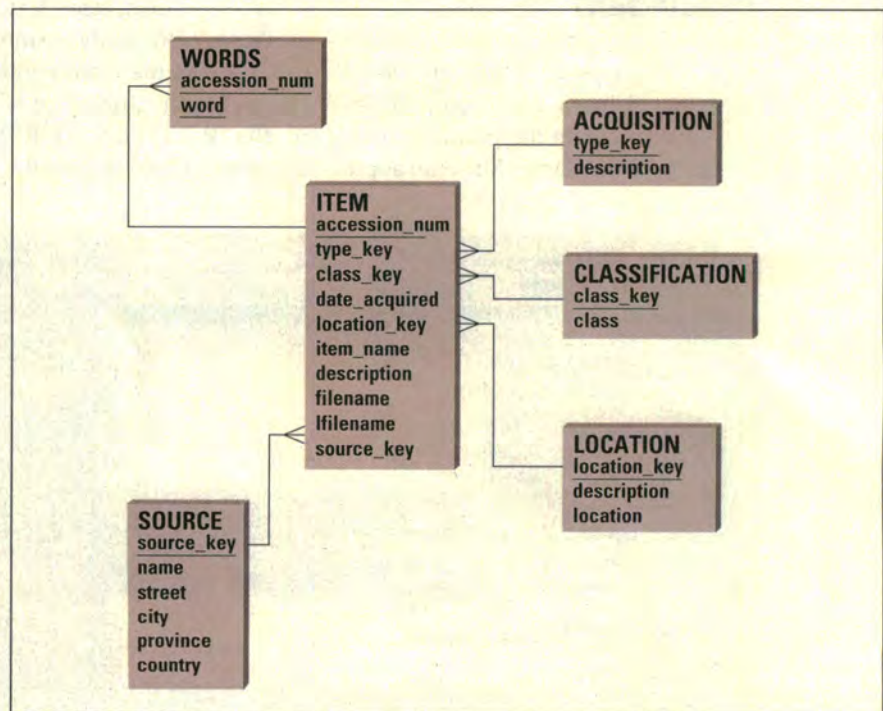


Figure 2. The database structure displayed in an entity-relationship diagram

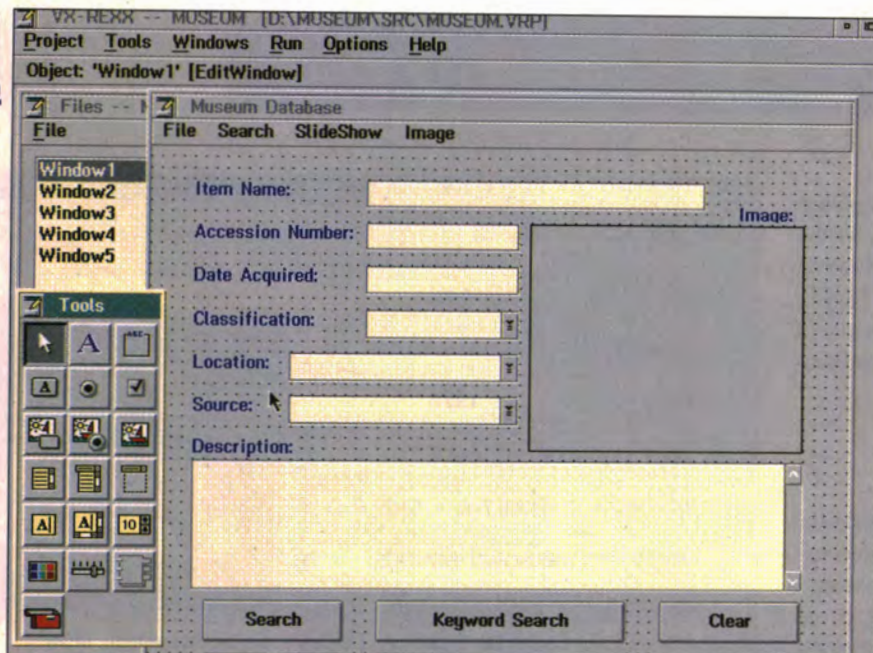


Figure 3. The main screen of the Games Museum database

the main table of information. One of these tables includes categories (that is, object, photograph, or book), and another includes donors. Figure 2 describes the Games Museum database structure in an entity-relationship diagram.

QUICK USER INTERFACE DEVELOPMENT

Developing a graphical user interface with a language such as C or C++ is complex. Using a visual tool, Uttley was able to create the graphical user interface for the Games Museum appli-

cation quickly and easily. The VX-REXX development environment makes controls available in a toolbar. These controls were added to the Games Museum database application by selecting them from the toolbar and dropping them onto application windows. Once placed on a window, each control's position, color, text, and other attributes were easily customized. Figure 3 shows the main window of the Games Museum database as it appears in the WATCOM VX-REXX visual development environment.

EVENT-DRIVEN PROGRAMMING

The user interface is connected to the application code by events. When a developer clicks on a button, for example, the button generates a click event. Event code is written in standard OS/2 REXX. VX-REXX's drag-and-drop programming allows the developer to write the event routines without typing code. The developer simply drags and drops controls from the application window into the event routine, then selects the desired action from a list. Figure 4 illustrates this drag-and-drop programming feature.

The code for each event can be edited separately, or all the code can be edited at once. Editing all of the code at once is useful for making global changes or for getting the big picture of an application. Developers also have the choice of using the built-in, VX-REXX editor or configuring the system to use an external editor.

CALLING DB2/2

DB2/2 includes an application programming interface (API) for REXX. An application, such as DB2/2, that has a REXX API provides routines a REXX program can call to interact with that application. Because VX-REXX is integrated with OS/2 REXX, it can be used with any REXX API. In addition, some applications are REXX-aware. Users can call macros written in REXX from

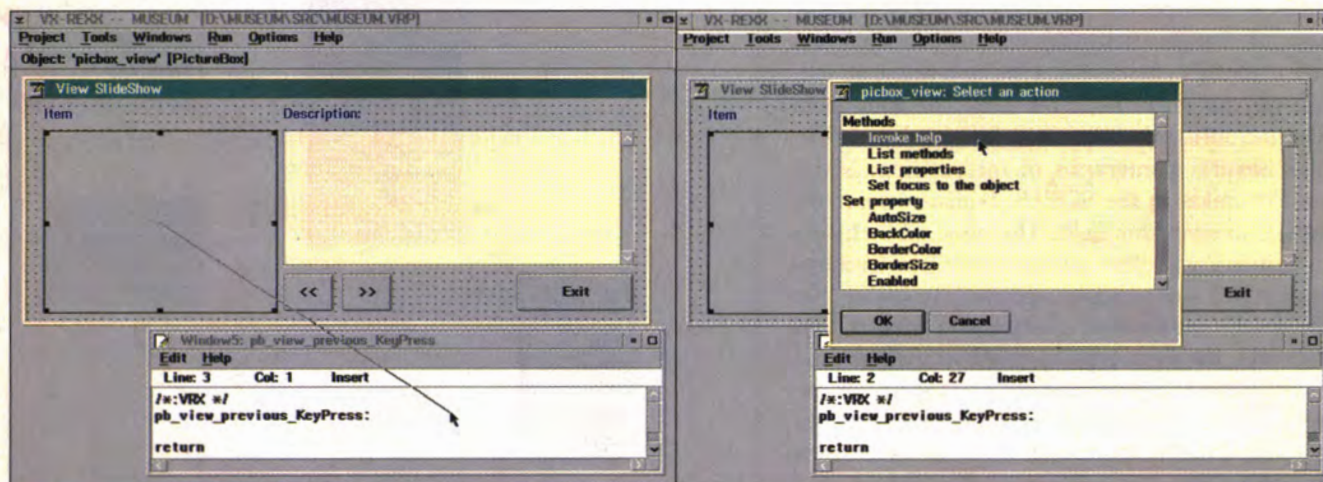


Figure 4. Drag-and-drop programming can eliminate the need to write code

within the application. The OS/2 Enhanced Editor and Lotus's AmiPro are two examples. VX-REXX can be used with this type of application to create visual macros.

Using DB2/2 involves:

1. Registering the DB2/2 routines
2. Starting DB2/2
3. Declaring a cursor
4. Issuing a SQL statement
5. Displaying the data
6. Closing the cursor
7. Stopping DB2/2.

Many of these actions involve using two DB2/2 routines. `SQLEXEC` is used to process all SQL requests. `SQLDBS` is used to call DB2/2 environment and utility routines.

Issuing an SQL statement and displaying the data are described in the following section; more information about using DB2/2 with REXX is described in the *WATCOM VX-REXX for OS/2 Programmer's Guide and Reference* and in the *Database 2 OS/2 Programming Reference*.

USING SQL

The Games Museum database application communicates with the DB2/2 database through SQL statements embedded in REXX functions. For example, when the right arrow button is pushed on the application's main window, the embedded SQL statement, `FETCH`, is issued to retrieve and display the fields of the next record in the database. In Figure 5, a code sample from the click event routine of the `Next` button provides a behind-the-scenes look at how SQL is used. In the sample, the call to the `SQLExec` routine finds the next record. The following three calls to `VRSet` display the values of the three variables that were found.

CREATING A RANKED TEXT SEARCH

The Games Museum database application features a ranked text search on the description fields of each record. The ranked text search, called a "keyword search" in the application, requests a word or series of words from

the user to form a search pattern. The resulting matches are ranked by how well they match the original search words. The order of the search results does not reflect the order in which the words appear in the description; the results are arranged in descending order by the number of words in the text string that satisfy the search query.

Before the search, certain words that have no important meaning of their own are eliminated, such as *it*, *or*, *and*, *the*, and *that*. Characters such as brackets and punctuation marks are also removed.

The search runs very quickly, because the database includes a search word index table called "Words," as shown in Figure 6. This index contains the description text broken into individual words. Stop words and unnecessary characters are not included. A separate, VX-REXX program compiles the word index as part of the update procedure.

Suppose you want to search the Games Museum database for the string "An early American game board." The search procedure would issue the SQL statement in Figure 6. The `SELECT` statement causes the database to return three fields: the accession number, item name, and number of words matched. Only descriptions containing at least one word from the search list will be selected. The `ORDER BY` clause causes the rows to be returned from the database in decreasing order of number of matches, thus ranking the matches.

TAPPING INTO OS/2 MULTIMEDIA CAPABILITIES

The Games Museum database application stores full-color images of museum pieces for many of the museum records. Like the Games Museum database application, other applications can benefit from this capability. Since a picture is worth a thousand words, pictures can replace descriptive text using only one field. They also accurately reflect the attributes of items and aid the usability of a database. Finally, users can use pictures for presentations and other external uses. For example, the Games Museum database applica-

```
/*:VRX*/
PBM_Click:

call SQLExec 'FETCH C1 INTO:ITEM_NAME, :ACCESSION_NUM, :DATE_ACQUIRED'
if SQLCA.SQLCODE \=0 then do
    call VRSet 'PBN', 'Enabled', 0
end

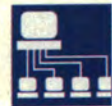
call VRSet 'EF_Item', 'Value', ITEM_NAME
call VRSet 'EF_Accession', 'Value', ACCESSION_NUM
call VRSet 'EF_Date', 'Value', DATE_ACQUIRED

return
```

Figure 5. Code sample from the click event

```
SELECT words.accession_num, COUNT(DISTINCT Word), item.item_name
FROM words, item
Where Word IN ('EARLY', 'AMERICAN', 'GAME', 'BOARD')
and words.accession_num = item.accession_num
GROUP BY words.accession_num, item.item_name
ORDER BY 2 DESC
```

Figure 6. SQL statement issued by search procedure





tion provides a slide show feature. The slide show consists of one or more item pictures and their attached descriptions. The slide show can be loaded on a disk and sent to customer sites to provide education and information. The slide show presentation is quick to develop, paperless, easy-to-use, and inexpensive. It can reduce the need for personal, on-site visits.

The museum pictures are stored in the database as digitized images. To create them, 35mm slides were scanned by a Nikon LS3510/AF. The resulting high-resolution files were converted to 256-color bitmap files using Image Alchemy, a conversion program. The program Wingif was then used to dither the files into a 16-color format that would allow low-resolution monitors to display the images.

The VX-REXX picture-box control was used to draw a box that would contain the images. At run time, assigning the path of the bitmap file to the picture box's `PicturePath` property causes the picture to be displayed.

Avedon also sees the potential of tapping into further multimedia capabilities available from WATCOM VX-REXX. The Museum carries many games that involve

sound and motion. To accurately depict these games and how they are played, the database could be modified to support audio and motion picture fields. In fact, these multimedia records could play on separate, simultaneous threads for added convenience and interaction.

THE FUTURE OF THE GAMES MUSEUM DATABASE PROTOTYPE

Avedon expressed enthusiasm for the future of the Games Museum database after reviewing Jennifer Uttley's Games Museum database prototype. The prototype allows him to present the proposed, database alternative to colleagues in a visual and realistic manner.

As for the final production version, issues regarding the transition of a hierarchical database to a relational database

remain in debate. However, the ability to store images, a graphical user interface front end, easy maintenance of the information, sophisticated, word search features, and usability of the database were all major improvements over the mainframe version of the database.

ACKNOWLEDGMENTS

I would like to thank the people who contributed to and reviewed this article, including Jennifer Uttley, Elliott Avedon, Rob Vietch, and Terry Stepien.

Grace Welch is a marketing specialist for Watcom, Waterloo, Ont., Canada, where she works closely with the C, C++, FORTRAN 77, Watcp, SQL, and VX-REXX product lines for OS/2. She recently received a B.A. from the University of Waterloo.

REFERENCES

Database 2 OS/2 Programming Reference, International Business Machines Corporation, First Edition, March 1993.

WATCOM VX-REXX for OS/2 Programmer's Guide and Reference, WATCOM International Corporation, 1993.



This article provides an in-depth walk-through of debugging routines using the REXX programming language's trace command. Programs can be tested entirely inside REXX without the use of external debugging tools.

BY STEVE CLARK

Debugging REXX Applications

In *The REXX Language: A Practical Approach to Programming*, author Michael Cowlshaw states that the single design objective of REXX was "to make programming easier than it was before." The simplicity of REXX allows programmers to deal more directly with the logic of a program.

The same design objective was applied to debugging REXX code. The REXX language includes many debugging features to make testing and debugging easier. Understanding where and how these debugging aids are used can improve the quality of your code and the efficiency of the development, testing, debugging, and documentation process.

Most of the information and examples here are appropriate for all REXX operating environments. Specific OS/2 features are clearly noted.

DEFINING TRACE ACTION

The REXX language processor offers many trace actions for use in debugging. Setting a trace action determines the amount of trace information to be displayed and whether the trace is run interactively. This can be controlled within a REXX program by the TRACE instruction or the built-in TRACE() function. These share most of the same parameters and can be used interchangeably in many situations.

The trace action is defined by a single parameter made up of either a whole number or an alphabetic character(word) option. The number trace parameter is only valid during interactive tracing. The most commonly used alphabetic character(word) options are:

- **Normal.** Failed commands are traced after execution. This is the default setting.

- **Off.** All tracing is turned off.
- **Results.** All clauses are traced before execution and results are displayed. This is the most common option.
- **Intermediates.** This option is similar to Results, but intermediate results are also displayed.
- **Labels.** All labels are traced as they are passed.

REXX recognizes an option by the first letter of its name so it is not necessary to type the full name. The alphabetic character(word) option may also include one or more question mark (?) prefix characters. The question mark prefix acts as a switch turning interactive tracing on and off. Examples of setting the REXX trace action using the trace instruction are shown in Figure 1.

REXX trace parameters are described in detail in the *OS/2 2.0 Procedures Language 2/REXX* reference manual.

DIFFERENCES BETWEEN TRACE AND TRACE()

If the TRACE instruction and the TRACE() function are so similar, why is it necessary to have both? In practice, the TRACE instruction is sufficient for most debugging tasks, and its simpler syntax is quicker to use. There are differences, however, between the two commands. The TRACE() function has some unique features that make it a necessary and valuable debugging aid.

The most obvious difference between TRACE and TRACE() is that the TRACE() function returns a result. The result is the current TRACE action. You can invoke the TRACE() function directly within an expression, and the result will be evaluated as part



Steve Clark

of the expression. Alternatively, the TRACE() function can be invoked with the CALL instruction and the returned result will be assigned to the special variable RESULT. Examples of calling the TRACE() function are shown in Figure 2.

Also, like other functions, TRACE() evaluates its option as an expression, whereas the TRACE instruction evaluates its option as a constant. The TRACE instruction is defined this way to simplify the syntax of the command and avoid surprises to the programmer. The instructions TRACE I or TRACE OFF will always set the expected trace action. However, using CALL TRACE I or CALL TRACE OFF may generate some unexpected trace behavior if the symbols i or off have been assigned values. Figure 3 illustrates that it is necessary to be explicit when using the TRACE() function and delimit the option with quotes.

The intent in Figure 3 is to set the trace action to Intermediates. Instead of doing so, the language processor evaluates the variable i, which has been incremented to 11 by the do i=1 to 10 loop. What is actually executed is CALL TRACE 11.

Interactive trace allows programmers to monitor program execution, change variables, reexecute clauses, execute program routines, and even alter program flow.

This example points out another difference. The number parameter, while valid for the TRACE instruction, is not valid for the TRACE() function. Using the number parameter with a call to the TRACE() function will cause a syntax error.

The less obvious differences between the TRACE instruction and the TRACE() function become apparent during interactive tracing; during interactive trace, the REXX processor pauses after most clauses for input from the keyboard. One exception is the TRACE

```
TRACE OFF      /* Trace off          */
TRACE 0        /* Trace off          */
TRACE ?R       /* Trace results - interactive */
TRACE I        /* Trace intermediate results */
```

Figure 1. Sample TRACE instructions

```
trace_value = TRACE()      /* Get trace option          */
trace_value = TRACE('R')  /* Get option and set results */
CALL TRACE ' ?R'           /* Set results - interactive */
```

Figure 2. Sample TRACE() function calls

instruction. Following a TRACE instruction, the language processor will continue to execute until the next pause. This is true whether the TRACE instruction is in the file or entered from the

TRACE OUTPUT

As each clause is executed, the trace output is displayed to the user. The display contains the executed source line number and text. A special symbol is also displayed to indicate the type of data being displayed. The ** symbol indicates that the displayed data is the source text of the executed clause. The +++ symbol indicates a trace message from the language processor.

When results are displayed, they are indicated by symbols that begin and end with the greater than sign (>). These symbols are shown in Figure 5.

INTERACTIVE TRACE

Interactive trace allows programmers to monitor program execution, change variables, reexecute clauses, execute program routines, and even alter program flow. The REXX processor pauses following the execution of most clauses and allows programmers to enter instructions from the console. In fact, all REXX instructions may be entered from the console during interactive trace.

In addition, there are two special instructions that control execution dur-

keyboard. In contrast, the language processor pauses after the TRACE() function, as it does for all functions.

Another difference is that during interactive trace, trace control is transferred to the console. All TRACE instructions in the program file are ignored. Only keyboard TRACE instructions are obeyed until interactive trace is switched off. In contrast, the TRACE() function is never ignored. The differences between TRACE and TRACE() are summarized in Figure 4.


```
do i=1 to 10
.
.
end
call trace i
```

Figure 3. Invalid TRACE() function call

ing interactive trace. These two instructions are the null line and the equal sign (=). Entering a null line by pressing the enter key will cause the processor to execute the next clause and continue executing until it reaches the next pause.

Entering = will reexecute the last clause traced in the current routine. This is an extremely useful command when used in conjunction with the available REXX instructions. If you are using the Results trace action and come

across a complex line of code, you can switch the tracing action to *Intermediates* by executing the TRACE() function from the keyboard and then reexecuting the clause. The TRACE() function is used so the language processor will pause and not continue to the next clause.

Figure 6 demonstrates this scenario. In the first execution of the clause, you can see that the variable *x* was compared to the result of the VERIFY() function and the result of that comparison is 0 or false. But since the values of the variables in this clause are unknown, it is difficult to see how the result was determined. So the trace action is changed with the TRACE() function and reexecuted to get more detail. The Results symbols now show that the variable (>V>) 'x' equals 1, string contains a date, delim_list contains a list of delimiters, and that there is a literal (>L>)

string *M*. The VERIFY function(>F>) has returned 3, the comparison operation (>D>) returned 0, and the final result is 0.

During an interactive trace, variables can be examined with the SAY instruction. The semicolon can be used to enter multiple clauses on a line. In fact, the processor requires that all instructions such as DO...END and IF...THEN be complete when entered, as in Figure 7.

While monitoring the program execution and examining the variables with the SAY instruction or through trace results, changing a variable's value is invariably useful. Variables can be modified with an ASSIGNMENT instruction, as in Figure 8.

Program subroutines can be executed directly from the keyboard with the CALL instruction. Subroutines can also be exited at any point with the

DON'T MISS OUT!

Now you can get Special Reports for *Virtual Reality*, *Neural Network*, and *AI in Finance* at incredibly **low** prices! This offer won't last forever, so order **now**. Don't take a chance!

Don't miss this incredible offer!



☒ **YES!** Please send me the Special Reports that I have marked below:

# of copies	1992 Virtual Reality Special Report \$9.95	Name _____
_____	March 1993 Virtual Reality Special Report \$9.95	Company _____
_____	August 1993 Virtual Reality Special Report \$9.95	Address _____
_____	1992 Neural Network Special Report \$7.95	City _____
_____	1993 Neural Network Special Report \$7.95	State _____ Zip _____
_____	Neural Net Primer Special Report \$7.95 (Available in December 1993)	
_____	AI in Finance Special Report \$7.95	Prepayment is Required.

Subtotal for _____ copies of all VRSR @ \$9.95 each
Subtotal for _____ copies of all NNSR @ \$7.95 each
Subtotal for _____ copies of AIFSR @ \$7.95 each
TOTAL

FAX (415) 905-2233 or call (800) 444-4881

Allow up to 6 weeks for delivery of your first issue. Payable in U.S. dollars. Please make checks payable to Miller Freeman. Canadian GST #124513185

New!

dbfREXX

READ AND WRITE dBASE FILES FROM REXX!



Now you can have simple database management from your REXX programs, without spending a lot of money! dbfREXX extends the REXX language with functions to read and write database files and indexes. Files supported are: DBF, DBT, & NDX. Add dbfREXX to your REXX programming arsenal.

SPECIAL!
\$ 59
thru 7/31/94

Access dBASE files from your C and C++ programs too!
Ask about dbfLIB Programmer's Library

(713) 537-0318

Visa and MasterCard accepted.



dSOFT Development Inc.
4710 Innsbruck Drive
Houston, TX 77066

RETURN instruction. This means that a programmer can dramatically alter program flow.

Program execution can also be halted with the EXIT instruction, although it is better to exit by calling the program's exit subroutine (for example, CALL EXIT_PROG). The exit subroutine would have been coded to perform any necessary cleanup, such as dropping queues or deleting temporary work files.

CONTROLLING INTERACTIVE PAUSES

Two trace options control pausing during interactive tracing. One of these is the LABELS option. This option changes the trace action to only trace and pause at label clauses. This is useful for limiting the amount of trace output and quickly progressing to the next subroutine of interest. Once a desired subroutine is reached, changing the trace action to Results or Intermediates will resume tracing of all clauses.

TRACE L is also useful when examining unfamiliar programs. Since only labels are traced, it can be used to generate an outline of the program flow showing the order in which each subroutine is executed.

The other trace option used for controlling interactive pauses is the number option. When a positive whole number is used as a trace option, the designated number of pauses will be ignored. Using a negative whole number has the same effect on pausing, but trace output is suspended for the designated number of pauses.

The number option is most useful for debugging loops. After stepping through a loop once, the number option can be used to initiate a full execution of the loop without pausing.

INTERACTIVE TRACE OF SUBROUTINES

When a subroutine is to be traced, the programmer can start interactive tracing by adding a TRACE instruction to the source either within the subroutine or prior to the call to the subroutine. In either case, the subroutine will be

	TRACE	TRACE()
Alter trace action from keyboard	Y	Y
Result returned	N	Y
Option is taken as a constant	Y	N
Pause after TRACE clause	N*	Y*
Ignored in file during interactive tracing	Y	N
Accept number option	Y	N
No parameter sets default trace action	Y	N

* Note: The pause behavior of the TRACE instruction and the TRACE() function when entered at the keyboard during interactive trace is not consistent in all implementations or versions of REXX.

Figure 4. TRACE vs. TRACE() behavior

When the trace action is set to 'Results' with the 'R' trace option, the results are preceded by the following symbols:

- >>> Results of an expression including assignments made with the parse instruction.
- >.> Value assigned to the period ('.') placeholder during parsing.

When the trace action is set to display 'Intermediates' with the 'I' trace option, then the following additional result symbols are used to identify the intermediate results:

- >C> Resolution of a compound symbol (symbols which contain a period). The value shown is not the value assigned the symbol but the stem portion of the symbol and the resolved components of the tail. Compound symbols are described in depth in the OS/2 2.0 Procedures Language 2/REXX Reference manual.
- >F> Result of a function call.
- >L> Literal.
- >O> Result of an operation on two terms. Operations include arithmetic, comparison and concatenation.
- >P> Result of a prefix operation (+ or -).
- >V> Contents of a variable.

Figure 5. Symbols used in trace output


```

9 ** If x = verify(string, delim_list, 'M');
>>> "0"

Enter: CALL TRACE 'I'
Enter: =

9 ** If x = verify(string, delim_list, 'M');
>V> "1"
>V> "01/17/92"
>V> "/- : "
>L> "M"
>F> "3"
>O> "0"
>>> "0"

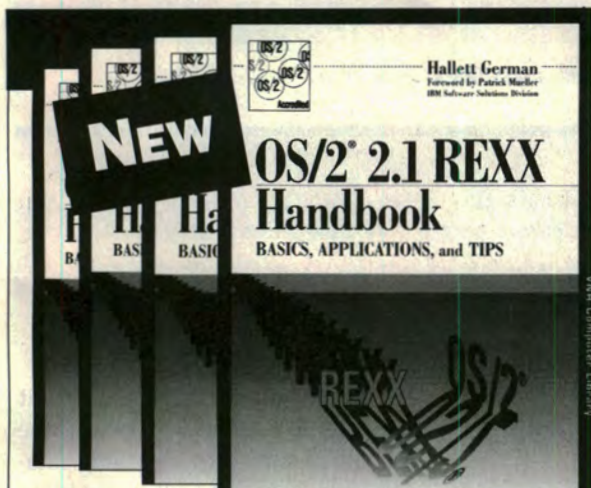
```

Figure 6. Reexecuting a clause to get intermediate results

traced. The difference is the programmer's control prior to and following the subroutine. Both approaches have their advantages.

Inserting a trace instruction within a subroutine will cause tracing to begin after entering the subroutine. There is no need to change the trace action when the subroutine ends. That happens automatically. The trace action of the caller is saved, and when control returns to the caller, that trace action is restored.

If a subroutine is called from numerous points in a program, it is not necessary to locate each call and add a TRACE instruction. Instead, a single TRACE instruction within the subroutine will turn tracing on each time the subroutine is executed. However, because tracing does not begin until after entry into the subroutine, it is not apparent where the call to the subroutine was made or what the source of the subroutine's parameters were. Fortunately,



OS/2 2.1 REXX Handbook Basics, Applications, and Tips

by Hallett German

"A great reference book; contains countless info not found in other REXX books."—Luc Monette. The most thorough and useful guide yet published on the use of REXX interpreters and compilers, in OS/2 and across other platforms. Includes a complete REXX tutorial, specific techniques for applications development, and reviews of third-party and shareware interfaces with OS/2 REXX.

Order from VNR (800-842-3636)
or from IBM (800-568-2694) IBM #SR-5250.



VAN NOSTRAND REINHOLD

MAIL ORDER DEPARTMENT

P.O. Box 6904, FLORENCE, KY 41022-9949

☐ **YES!** Please rush me the new **OS/2 2.1 REXX Handbook Basics, Applications, and Tips** (0-442-01734-0) to examine FREE for 15 days. At the end of that time, I'll send \$29.95, plus local sales tax, and a small charge for shipping/handling, or simply return the book and OWE NOTHING.

☐ **SAVE MONEY:** Check here if enclosing cash, check, money order or credit card information with order. Publisher will pay shipping/handling charges on all prepaid orders. Our 15-day return/refund guarantee applies. Local sales tax must be included.

Name

Company

Address

City State Zip

Telephone No.

Please charge my credit card. VNR pays shipping/handling. If I return the book(s) at the end of 15 days, charges will be cancelled.

☐ Visa ☐ Mastercard ☐ American Express

Card No. Exp. Date

SIGNATURE

(Offer invalid without signature)

Offer good in U.S. and territorial possessions only and subject to credit department approval. Prices subject to change and slightly higher in Canada. No shipment to P.O. boxes without prepayment.
8/94 1637



REXX provides a special variable and a built-in function for retrieving this information. The special variable SIGL contains the line number of the clause that transferred control to the subroutine, as shown in Figure 9. The built-in function SOURCELINE() used with the special variable SIGL will return the exact text of the clause that called the active subroutine.

Adding a trace instruction prior to the call to a subroutine establishes the trace action prior to, during, and after

By modifying the variables that make up the calling parameters and repeatedly calling a subroutine, subroutines can be fully tested in a single execution of the program.

the execution of the subroutine. This allows a programmer to see exactly where the call takes place. It provides an opportunity to see and modify the subroutine's calling parameters and to see and modify the result when the subroutine returns. It also allows the programmer to easily reexecute the subroutine.

Since the REXX processor does not pause following a trace instruction, it is recommended that a NOP instruction be added following the TRACE instruction. The language processor will pause after the NOP instruction, which provides an opportunity to examine and possibly modify variables before entering the subroutine. The DEBUG_RX.CMD program illustrates this with the clause TRACE ?R; NOP just before the call to the DIFF subroutine. The TRACE instruction is

```
Enter: say 'The value of x is' x
       The value of x is 34

Enter: do i=0 to var.0; say 'Var.'i 'is' var.i; end
       Var.0 is 3
       Var.1 is 25
       Var.2 is 12
       Var.3 is 58
```

Figure 7. Examining variables with SAY

```
Example:

6 ** If x = y;
>>> "0"                                /* Result indicates false */

Enter:  x=y                               /* Modify x with assignment */
Enter:  =                                 /* Re-execute */

6 ** If x = y;
>>> "1"                                /* Result is now true */
```

Figure 8. Altering program flow with an ASSIGNMENT instruction

```
Enter: say sigl sourceLine(sigl)
       126 call subr1 parm_a parm_b /* Line which called subr1 */
```

Figure 9. Displaying a subroutines caller

used instead of the TRACE() function because it is ignored if interactive trace is already switched on.

When the tracing action is set prior to calling a subroutine, that action will be in effect after the subroutine ends. The programmer can take advantage of this by using the equal sign (=) trace command to reexecute the call to the subroutine. The = reexecutes the last clause traced in the current routine. After control is returned to the caller by the RETURN instruction, the last clause traced in the current routine is the clause that called the subroutine. Therefore, entering = will reexecute the call to the subroutine, as shown in Figure 10.

This is an extremely efficient debugging technique. By modifying the variables that make up the calling parameters and repeatedly calling a subroutine in this manner, subroutines can be fully tested in a single execution of the program.

When using this technique, it may be desirable to quickly return to the calling routine and reexecute the subroutine. There are two ways to do this. Entering TRACE OFF will turn off tracing until the subroutine returns to the caller. Entering the RETURN instruction will change the program flow and force an immediate return to the caller. If the RETURN instruction is used in a subroutine called as a function, it must return


```

/* When DEBUG_RX.CMD is executed with the trace instruction */
/* suggested above, the trace scenario might proceed as shown here. */

12 ** Nop;
    +++ Interactive trace. "Trace Off" to end debug, ENTER to Continue.

Enter: say yr1 yr2 sz          /* Examine variables */
      00 10 2

Enter:                          /* Step through subroutine */

13 ** call diff yr1, yr2, sz;
24 ** Diff:                    /* Start of subroutine */

Enter: trace -6                /* Suspend trace for 6 lines */

32 **      ans = x - y;
    >>>      "-10"

33 **      If abs(ans) >= abs(( x + z) - y );
    >>>      "0"

34 **      If abs(ans) >= abs(x - ( y + z) );
    >>>      "0"

35 **      Return rc;          /* Return to caller */
    >>>      "-10"
    >>>      "-10"

Enter: yr1 = 50                /* Change calling parameter */
Enter: =                      /* Re-execute subroutine */

13 ** call diff yr1, yr2, sz;
24 ** Diff:                    /* Start of subroutine */

```

Figure 10. Reexecuting a subroutine in interactive trace

a result, such as RETURN 0, or a syntax error will be generated.

Once a tracing action is set, it will apply to any called subroutines. If there is no interest in tracing a subroutine, entering TRACE OFF will switch off tracing until that subroutine ends. But if a program contains a subroutine that is called repeatedly, such as a logging or messaging routine, it can be an annoyance to constantly be switching off tracing from the keyboard. In this instance, it is preferable to programmatically switch trace off with a clause

in the offending subroutine. TRACE instructions in the program cannot be used for this purpose because they are ignored during interactive trace. But the TRACE() function can be added to a subroutine to programmatically switch tracing off.

USING EXTERNAL ROUTINES

Reexecuting subroutine calls with the equal sign (=) trace command works very well with external routines. Whereas an internal routine cannot be changed and retested without ending

and restarting the program, an external routine can be modified between calls without ending the calling program. During development, it can be very efficient to write and test many routines externally. Once testing is completed, they can be converted to internal routines. If the file name of the external routine is kept the same as the label, then the only conversion is to copy the routine into the program.

In determining if this development technique is appropriate, keep in mind that there are no external variables in REXX. All of the caller's variables are hidden from an external routine just as

*During
development, it can
be very efficient to
write and test many
routines externally.*

if a PROCEDURE instruction were used. The DIFF subroutine in the DEBUG_RX.CMD sample program is an example of a routine that can be used unchanged either as an internal routine or as an external REXX routine. DIFF does not use any variables from the caller's environment. It receives data through the calling arguments and returns data to the program as a result. To convert DIFF to an external routine, simply copy the routine into a file named DIFF.CMD and delete it from the program.

SAVING TRACE OUTPUT

In the OS/2 operating environment, the REXX processor generally uses the default input and output streams STDIN and STDOUT to read from the keyboard and write to the display. Trace output, on the other hand, is written to the STDERR output stream. Thus it is very easy to save or print trace output by

executing the program with STDERR redirected to a file or a printer. The language processor detects when STDERR has been redirected and will not allow interactive trace to be switched on. When using redirection, STDERR is referenced by the internal file ID (handle) 2. When entered at the OS/2 command prompt, this example will execute the program MYAPP and redirect any trace output to the file TRACE.OUT:

MYAPP 2>TRACE.OUT

The OS/2 PMREXX utility allows execution and debugging of REXX programs within a Presentation Manager window. It can also be used to save trace output. Among the many features of PMREXX are scrolling and saving out-

With the commands available in REXX, it is simple to automate running test cases that will test the complete program or specific routines within the program.

put, selective copying to the clipboard, pasting to input, and switching interactive trace on and off with the mouse.

PMREXX is an optional system utility in OS/2 2.0 and 2.1. If not originally installed, it can be installed from the OS/2 desktop by selecting OS/2 System, System Setup, Selective Install, System Configuration (click on 'OK'), OS/2 Setup and Installation, Optional System Utilities in OS/2 Installation, and PMREXX.

```

/*****
** REVERSE.CMD
** Filter to reverse lines in a file
** - Uses standard piping/redirection
*****/

trace_value = trace('OFF')           /* interactive trace */
if trace_value = '?R' then trash = trace('R') /* must be off */

do while lines('STDIN')
  line = linein('STDIN')             /* read line from STDIN */
  line = reverse(line)               /* reverse line */
  call lineout 'STDOUT', line        /* write line to STDOUT */
end

```

Figure 11. Resetting trace action in a filter

USING RXTRACE ENVIRONMENT VARIABLE

The REXX processor provides an external means of switching on interactive trace in the OS/2 operating environment. If the environment variable RXTRACE is set to ON, the language processor will automatically set the trace action to ?R. It is, therefore, not necessary to modify a REXX source file to switch on interactive trace. The RXTRACE environment variable is most helpful for remotely debugging user problems that cannot be duplicated. The user can easily provide the trace information needed by switching trace on with SET RXTRACE=ON, executing the program with the trace output (STDERR) redirected to a file or printer, and then sending the output to the programmer.

Using the RXTRACE environment variable works quite well unless the ?R trace action is not the preferred or appropriate setting. For instance, ?I might be preferred because it provides more detailed trace information. Or, if the program is a filter, then interactive trace is not appropriate because the language processor will try and interpret the STDIN input stream as REXX instructions. In these cases, some additional code can correctly reset the trace action.

The REVERSE.CMD sample program demonstrates how to program-

matically test and reset the trace action. REVERSE.CMD is a simple filter that takes lines from STDIN, reverses them with the REVERSE() built-in function and then writes to STDOUT. Because it is a filter, REVERSE.CMD will not execute correctly in interactive trace mode. If interactive trace has been switched on using the RXTRACE environment variable, the first two lines of the program will reset the trace action. The first line calls the TRACE() function to query the current trace action and immediately turn tracing off. The current trace action is stored in the TRACE_VALUE variable. The second line checks TRACE_VALUE; if it was set to ?R, the trace action is reset to R, as shown in Figure 11.

AUTOMATING TEST CASES

An important part of application development and maintenance is verifying that a program works as designed in the original release and in successive generations of enhancements and maintenance. This validation is done with test cases. Two problems in running test cases are the time it takes to execute them and the difficulty of designing tests that fully validate all of the program's routines. With the commands available in REXX, it is simple to automate the running of test cases that will test the complete program or



```
*****
** DEBUG_RX.CMD program to demonstrate automated test cases
*****/
parse upper arg parms
if parms = 'TEST' then signal Run_Test_Cases

trace o
yr1 = 00
yr2 = 10
sz = 2

trace ?r; nop          /* trace DIFF subroutine */
call diff yr1, yr2, sz
ans = result
say yr1 '-' yr2 '=' ans

call exit

/*****
** DIFF() returns the difference between a pair of counters that
** are reset to zero after they reach maximum value.
** ex. last two digits of the year DIFF(92,00,2)) -> -8
*****/
Diff: procedure
  parse arg x, y, z .
  if datatype(x,'W') <> 1 ,
    | datatype(y,'W') <> 1 ,
    | datatype(z,'W') <> 1 then ans = ''
  else do
    z=10**z
    ans=x-y
    if abs(ans)>=abs((x+z)-y) then ans=(x+z)-y
    if abs(ans)>=abs(x-(y+z)) then ans=x-(y+z)
  end
  return ans

/*****
** Read and execute test cases
*****/
Run_Test_Cases:
  TC_input = 'DEBUG_RX.TST'
  TC_test_case = ''
  do while lines(TC_input)>0
    TC_Test_case = linein(TC_input)
    interpret TC_test_case
    call trace 'OFF'
  end
  call exit

/*****
** Exit
*****/
Exit:
  exit
```

Figure 12. Program that reads and executes its own test cases

specific routines within the program.

It was discussed earlier that testing can be done during interactive trace by entering REXX instructions from the keyboard. Those instructions are interpreted immediately by the language processor. The INTERPRET instruction will also process REXX instructions. Creative use of the INTERPRET instruction allows programmers to exercise control, similar to interactive trace, from outside the program.

The DEBUG_RX.CMD program, shown in Figure 12, demonstrates how to take instructions from a file of test cases and execute them with the INTERPRET instruction. The RUN_TEST_CASES routine in DEBUG_RX.CMD provides this function. If the program is started with a parameter of TEST, then the SIGNAL instruction transfers control to the RUN_TEST_CASES routine, which reads the DEBUG_RX.TST file. As each line is read, it is processed by the INTERPRET instruction.

DEBUG_RX.TST, shown in Figure 13, contains instructions that test the DIFF() routine in DEBUG_RX.CMD. This file looks similar to a REXX file because it contains REXX instructions; however, each line represents a literal string that will be processed by the INTERPRET instruction. INTERPRET requires that all instructions, such as DO...END and IF...THEN, be complete. Therefore, instructions and comments in the test case file cannot span multiple lines. Long instructions are pieced together in multiple assignment statements with a semicolon separating the clauses. The basic test scenario for DIFF() is assigned to the variable TEST in this manner. The test cases are run by changing the values of the variables used in TEST and giving the instruction INTERPRET TEST (RUN_TEST_CASES actually executes INTERPRET INTERPRET TEST). All messages are written to STDERR so they can be easily redirected to a file.

Incorporating this method of saving and running test cases into the development process improves the quality of the code produced, the efficiency of development, and the level of


```

/*****
/* DEBUG_RX.TST Test case file for DEBUG_RX.CMD
/*
*****/

call lineout STDERR, 'Beginning of test case file'
call lineout STDERR, ' '

call lineout STDERR, 'Test cases for DIFF() function'
call lineout STDERR, ' '

/* DIFF() returns the difference between a pair of counters that */
/* are reset to zero after they reach maximum value. */
/* ex. last two digits of the year DIFF(92,00,2)) -> -8 */

/* define test (note: all instructions must end with semi-colon) */
test = "call lineout STDERR, 'DIFF('yr1','yr2','sz')';"
test = test "ans = diff(yr1, yr2, sz);"
test = test "if ans == expect then call lineout STDERR, ans '-> OK';"
test = test "else do;"
test = test " call lineout STDERR, 'Test case failed';"
test = test " call lineout STDERR, 'Expected result is:' expect;"
test = test " call lineout STDERR, 'Actual result is: ' ans;"
test = test " call exit;"
test = test "end;"
test = test "call lineout STDERR, ' ';"

/* run test cases */
yr1 = 10; yr2 = 00; sz = 2; expect = 10
interpret test

yr1 = 90; yr2 = 00; sz = 2; expect = -10
interpret test

yr1 = 49; yr2 = 00; sz = 2; expect = 49
interpret test

say
call lineout STDERR, 'All test cases completed successfully'
call lineout STDERR, 'End of test case file'

```

Figure 13. Sample test cases for DEBUG_RX.CMD

documentation. The quality of the code is improved because there is a record of the actual test cases that have been run. The efficiency of development improves because test cases are captured as each function is developed rather than as an afterthought once coding is complete. The level of documentation improves because the test cases illustrate exactly what a piece of code is expected to do.

SUMMARY

Debugging REXX programs has a number of similarities with other languages and debug utilities. Adding a trace command into a REXX program is equivalent to setting a breakpoint into other languages. Other languages or utilities may also have the ability to examine variables, set variables, and even step through program execution interactively. In addition, REXX has a number of unique debugging features such as the ability to change the trace action, execute instructions, and call subroutines interactively. It can also reexecute a line of code or a subroutine. These features combine to create a very efficient debugging environment.

Steve Clark, ISSC Corp., Internal Zip WC7D, P.O. Box 2150, Atlanta, Ga. 30301-2150. Clark is a staff programmer in the Programmable Workstation Applications Development department at the ISSC Solution Center, Atlanta. He joined IBM in 1979 and has worked in application development in IBM and ISSC since 1984. He has a B.A. in accounting from Georgia State University.

REFERENCES

Cowlshaw, M.F. *The REXX Language: A Practical Approach to Programming*, Englewood Cliffs, N.J.: Prentice Hall, 1990. (IBM Doc. ZB35-5100)

OS/2 2.0 *Procedures Language 2/REXX Reference Manual* (IBM Doc. S10G-6268)



This article shows you how to add REXX support to your OS/2 applications with minimal fuss. All you need is a C/C++ compiler and the OS/2 Toolkit. By ERIC GIGUERE

Designing REXX-Aware Applications

These days, users demand a lot from the applications they use. One of the most useful features an application can have is the ability to customize its operation using a programming language of some kind. These programming languages are often referred to as macro languages to emphasize their special relationship with application programs. An application with a macro language appeals to expert users, who can tailor the application to suit their own needs, and system managers, who can use the macro capability to standardize and automate procedures for all users of the application.

Even if you can see the advantages to adding a macro capability to an application, as a developer you may be a bit leery about the amount of work involved. At first glance it hardly seems trivial: you have to design a programming language (or adapt an existing one) and implement an interpreter for the language. You may feel your time and energy is better spent refining other parts of your application—unless, of course, you use REXX as your macro language. This article shows you how to add REXX support to your OS/2 applications with a minimum of fuss. All you need is a C or C++ compiler and the OS/2 Toolkit. Full working examples of

the topics in this article are available electronically as well.

REXX AS A MACRO LANGUAGE

You are probably familiar with REXX as a stand-alone programming language, either as an alternative to the OS/2 batch language or as part of a development system. However, REXX was also designed for use as a macro language and has a well-documented API for access to the interpreter.

There are several advantages to using REXX as your application's macro language:

- REXX is a standard feature of OS/2 2.x.
- Using REXX saves you the trouble of designing your own language and interpreter. All you do is add application-specific features.
- For users, REXX is easy to learn. Books and on-line documentation are plentiful. If their favorite applications all use REXX as a macro language, they have less to learn.
- In Presentation Manager applications, tools like VX-REXX can be used to develop window-based macros.

REXX has its share of faults as well. Because it's interpreted, it may be slower than you'd like. It lacks proper support for complex data types and has unusual


```
rc = REXXStart( LONG numargs, PRXSTRING args, PSZ name, PRXSTRING instore,
                PSZ envname, LONG calltype, PRXSYSEXIT exitlist,
                PSHORT retcode, PRXSTRING result );
```

Figure 1. Prototype for the REXXStart function

scoping rules for variables. But its accessibility and ease-of-use far outweigh these problems, especially when used as a macro language, where short and simple programs are the norm.

REXX-AWARE APPLICATIONS

An application that uses REXX as its macro language is a REXX-aware application. Minimally, a REXX-aware application must meet two criteria. First, it must be able to run a user-defined REXX program within its own process space and preferably on a separate thread. Merely starting a new process that runs CMD.EXE to execute a REXX program is not enough—the application must call the interpreter directly. Second, the REXX macros must be able to access and control the application. This can be done using external functions, sub-command handlers, or system exits, which are all standard REXX facilities.

The Enhanced Editor (EPM) text editor is an example of a REXX-aware application. Start the editor and enter the following text:

```
/**/
do i = 1 to 10
  call ETKInsertText 'This is line' i
end
```

Save this file to disk as test.ern. Open the command dialogue by pressing Ctrl+I and type the command:

```
rx test.ern
```

EPM then calls the REXX interpreter to run the program you just saved to disk. That program in

turn calls back into EPM using the ETKInsertText function and inserts 10 lines into the current editing session.

REXX STRINGS

REXX is a string-based language. A special data structure is provided to handle strings because, unlike C, REXX strings can contain null characters. The RXSTRING data structure is defined as follows:

```
typedef struct {
    ULONG strlength;
    PCH strptr;
} RXSTRING;
```

The strlength field is the length of the string, and the strptr field points to the start of the string. The maximum length of a REXX string is the maximum value that can be stored in a ULONG, which for all practical uses is limitless. The RXSTRING data structure and the prototypes for all the REXX functions are defined in the REXXSAA.H header file, which you should include in your source files. (SAA stands for Systems Application Architecture. REXX is also known inside IBM as the SAA Procedures Language.) You will also have to link to the rexx.lib file, included with the OS/2 Toolkit.

There are some important rules to using RXSTRINGs properly. First, there is a difference between a null string and a zero-length string. The strptr field in a null string will be set to NULL and strlength will be zero. The strptr field in a zero-length string will be non-NULL, but strlength will be zero. In neither case should you try to access the value pointed to by

strptr. You will in most cases (but not always) treat both null strings and zero-length strings the same way. (For example, if passed as a function parameter, a null string means an argument was omitted.)

Second, treat all RXSTRINGs the REXX interpreter passes to your application as read-only data structures. The only exceptions are result strings; the interpreter will pass you an RXSTRING with strptr set to a pre-allocated 256-byte buffer and strlength set to 256. If your result string is 256 bytes or smaller, simply copy it into this buffer and set strlength to the appropriate value. If your result string will not fit in the buffer, you must allocate a new buffer using DosAllocMem (do not use your C library's malloc routine!), set the strptr field to point to this buffer, copy the data, and set the strlength field. Do not free the original strptr value.

Third, before passing any RXSTRINGs to one of the REXX functions, you must initialize the strptr and strlength fields appropriately. For result strings, you point strptr at your own data area and set strlength to the size of the data area. If the result string is too small to fit, REXX will allocate a new one with DosAllocMem and reset the value of strptr. After processing the result, it is your responsibility to free the new data area with DosFreeMem. (You can let REXX allocate the buffer for you every time by setting strptr to NULL, but this results in a call to DosAllocMem every time. Most result strings are short, so a 256-byte buffer is often adequate.)

For convenience, REXX will always null-terminate any RXSTRINGs it passes to external functions, sub-

command handlers or system exits, which are discussed later. You may use the normal C string functions to process these strings if embedded null characters are not expected.

THE REXXStart FUNCTION

Running a REXX macro is as easy as calling a C function, as shown in Figure 1. It looks complicated, but it's not.

- **numargs** is the number of arguments you are passing to the macro. Set it to 0 if no arguments are being passed.
- **args** points to the first element in the array of argument strings. If **numargs** is 0, you can set this to NULL.
- **name** is the name of the REXX program to execute. If no extension is provided, a default of **.CMD** is used. If an absolute path is not specified, the interpreter looks first in the current directory and then down the **PATH** to find the file.
- **instore** is used when storing macros in memory or in tokenized form. Set it to **NULL** for now.
- **envname** is the name of the default subcommand environment for the macro. Unless you are defining a subcommand handler, set it to **CMD**.
- **calltype** tells how the interpreter should be calling the macro, either as a command (**RXCOMMAND**),

subroutine (**RXSUBROUTINE**), or function (**RXFUNCTION**).

The call type determines how many arguments it should expect (commands usually have a single argument string; subroutines and functions may have multiple argument strings) and how they treat the result (subroutines do not return a result). The **PARSE SOURCE** instruction in REXX will return a different string for each call type as well. In most cases, you want to set this to **RXCOMMAND**.

- **exits** is an array of system exit definitions. Set it to **NULL** unless you are using system exits.
- **retcode** points to a **SHORT** where the interpreter will store the macro result if the result string can be converted to a 16-bit integer. (The documentation says **retcode** is a **PLONG**, but the function prototype correctly shows it as a **PSHORT**.)
- **result** points to an **RXSTRING** where the interpreter will store the macro result string.

RexxStart will not return until the REXX program has finished executing or an error occurs. If the result is zero, the macro executed successfully to termination. A non-zero value indicates an error of some kind, and these are documented on line.

And, if you're wondering, the interpreter can be safely called simultaneously by different threads. It is also safe to call the interpreter when processing an external function call or a subcommand request.

THE TYPICAL CALL

Ninety percent of all macro calls can be handled with the simple cover function shown in Figure 2, which starts a macro, passes it a single argument, and ignores the result value.

If you are interested in the result, simply add a **PRXSTRING** as a

```
#include <os2.h>
#define INCL_REXXSAA
#include <rexxsaa.h>

BOOL RunMacro( PSZ filename, PCH argstring, ULONG len )
{
    char    buf[ 256 ];
    RXSTRING arglist[1];
    APIRET  rc;
    RXSTRING result;
    SHORT   retcode;

    result.strptr    = buf;
    result.strlength = 256;

    arglist[0].strlength = len;
    arglist[0].strptr    = argstring;

    rc = RexxStart( 1, arglist,
filename, NULL, "CMD",
                                RXCOMMAND,
NULL, &retcode, &result );

    if( result.strptr != buf ){
        DosFreeMem( result.strptr );
    }

    return( rc == 0 );
}
```

Figure 2. Typical call to start a REXX program



parameter and pass it to `RexxStart` instead of result.

THREADS AND REXX

The `RexxStart` function can be called by any thread in your application. In fact, it's good practice to run any REXX macros on their own threads for at least two reasons:

1. The macro can be halted by the main thread with a call to the `RexxSetHalt` function.
2. In Presentation Manager programs, running a REXX macro on a message-enabled thread can lock up the system.

A message-enabled thread is a thread with a message queue. The main thread of your application is typically a message-enabled thread. If you call `RexxStart` on a message-enabled thread that has at least one window visible, the desktop may lock up while the REXX macro executes because the window's messages are not being processed. It may not be noticeable for short programs, but if the REXX macro does lengthy operations (a database fetch, for example), the user will notice and complain.

EXTERNAL FUNCTIONS

The simplest way for a REXX macro to communicate with your application is through the use of external functions. An external function is an entry point into your executable that the REXX interpreter will call when the REXX program invokes a function whose name you have registered with the interpreter. External functions can also be packaged into DLLs for use by any REXX program in the system. The difference is that external functions in an executable are only available to the process that registered them.

The entry point in your executable is simply a C function, but it must be declared as shown in Figure 3.

```
#define INCL_REXXSAA
#include <rexxsaa.h>

RexxFunctionHandler ExtFunc;

ULONG APIENTRY ExtFunc( PCHAR funcname, ULONG numargs,
                        PRXSTRING args, PSZ queueName, PRXSTRING result )
{
    .....
}
```

Figure 3. Prototype for an external function

Entry points do not need to be exported. Before starting any REXX macros, you register your entry points with the interpreter using the `RexxRegisterFunctionExe` function:

```
extern RexxFunctionHandler ExtFunc;
RexxRegisterFunctionExe(
    "MyFunction",          ExtFunc );
```

The name of the REXX function ("MyFunction") has no relation to the name of the C function (`ExtFunc`). You may register several external functions with the interpreter and have them use different entry points or have them all use the same entry point.

The interpreter will call one of your entry points whenever it encounters a registered function. The parameters it passes are:

- `funcname` is the name of the function being called. Check this value if you have registered several functions with the same entry point.
- `numargs` is the number of argument strings being passed to the function.
- `args` is the list of argument strings, an array of `RXSTRING` structures.
- `queueName` is the name of the currently defined external REXX data queue. You can ignore this parameter unless you are working with queues.

- `result` points to an `RXSTRING` where the function result string is to be stored.

The external function should return 0 if no errors occurred. The interpreter will return the result string to the program. If an error occurs, the function should return 40, which will cause a syntax error (incorrect call to routine) in the program. (This is the only level of error available.)

In Figure 4, there is a simple external function that returns the lowercase version of its single argument string. (Remember that when your function is called, REXX will pass a preallocated buffer for you to store the result string. Use this buffer whenever possible.)

Remember that the external function (or any entry point in your application) will be called on the REXX macro's thread. Be careful: most Presentation Manager functions require that the thread be message-enabled, which means you will have to create a message queue if you need to use such a function. This is safe to do providing the thread does not create any windows and immediately calls the `WinCancelShutdown` function to remove the message queue from the system shutdown sequence. You may need to use semaphores to control access to process-wide resources

if two or more threads require them.

SUBCOMMANDS

Another way for a REXX macro to talk to the application is through the use of subcommand handlers. The following example invokes the CMD subcommand handler to do a directory listing:

```
address cmd
'dir c:\*.*'
```

Expressions that are not understood by the REXX interpreter are passed to the current subcommand handler for processing. The ADDRESS instruction and ADDRESS function can be used to change and identify the current subcommand handler.

Like an external function, a subcommand handler is an entry point in your executable, but with different parameters, as shown in Figure 5.

You register the entry point using REXXRegisterSubcomExe:

```
extern REXXSubcomHandler MyHandler;
REXXRegisterSubcomExe( "FOO", MyHandler,
    NULL );
```

The first parameter is the name of the subcommand handler (the name of your executable is a good choice), the second is the address of the entry point, and the third is a pointer to eight bytes of user-defined data. It is unusual to register more than one subcommand handler in an application.

The interpreter calls your entry point when a subcommand must be handled. Following are the parameters it passes:

- Command is the subcommand string.
- Flags will be used to set the completion status of the subcommand.
- Result points to an RXSTRING where the subcommand result string is to be stored.

It is the subcommand handler's responsibility to parse the command string and act on its contents. When done, the handler must set the completion status (one of the constants RXSUBCOM_OK, RXSUBCOM_ERROR, or RXSUBCOM_FAILURE) and set the result string. The result string will be assigned to the RC variable.

A subcommand handler example is included in the example programs.

SYSTEM EXITS

A system exit is a way to hook into the interpreter and modify its behavior. When an event you've expressed interest in occurs, the interpreter will call an entry point in your application and allows you to handle the event yourself. At the present time, the REXX interpreter defines system exits for the following events:

- External function calls

```
REXXFunctionHandler Lower;

ULONG APIENTRY Lower( PCHAR funcname, ULONG numargs,
    PRXSTRING args, PSZ queueName,
    PRXSTRING result ) {
    PVOID mem = NULL;
    ULONG i;

    if( numargs != 1 ) return( 40 );

    /* Allocate new buffer if necessary */

    if( args[0].strlength > result->strlength ){
        if( DosAllocMem( &mem,
            args[0].strlength,
            PAG_COMMIT |
            PAG_READ |
            PAG_WRITE ) != 0 )
            return( 40 );

        result->strptr = mem;
    }

    /* Copy and convert... */

    for( i = 0; i < args[0].strlength; ++i
        ){ result->strptr[i] =
            tolower( args[0].strptr[i] );
        }

    result->strlength = args[0].strlength;

    return( 0 );
}
```

Figure 4. A simple external function

- Subcommands
- External data queue processing
- Standard input and output
- Halting and tracing testing
- Program initialization and termination.

Your application may choose to handle these events in its own fashion or let the interpreter handle them for you. For example, it is common for Presentation Manager applications to use the standard input and output exit to redirect SAY statements to their own output window.

Activating a system exit is a two-step procedure. First, each exit handler must be registered with the interpreter using `RexxRegisterExitExe`. Then a list of exits to call must be passed as one of the parameters to `RexxStart` when a macro is started. No exit will be called if it is not in this list. A system exit example is included in the example programs that can be downloaded from CompuServe.

THE VARIABLE POOL

When processing an external function, subcommand or system exit, the application can use the `RexxVariablePool` function to set and

```
#include <rexxsaa.h>

RexxSubcomHandler MyHandler;

ULONG APIENTRY MyHandler( PRXSTRING command, PUSHORT flags,
                          PRXSTRING result )
{
    .....
}
```

Figure 5. Example of parameters for a subcommand handler

get the values of variables in the REXX program. The variable pool can be accessed when one of your application's entry points is called by a REXX program. See the example programs for details.

EXAMPLE PROGRAMS

Example programs demonstrating the topics discussed here are available for you to download and incorporate into your own programs. One of the examples shows how all the pieces can be combined into a single application.

CONCLUSION

Many details have been left to the example programs due to space constraints, but this article has cov-

ered the important points. As you can see, adding REXX support to an application is not very hard. It's usually harder to decide what the interface should look like than actually implementing it!

Eric Giguere is a software developer with Watcom International and chief architect of the VX-REXX run-time system and its associated Object Development Kit. Eric has had extensive experience with both graphical user interface and REXX development, including Motif and Microsoft Windows, as well as OS/2.

Example programs are available on CompuServe's OS2DF2 Forum, OS/2 Developer section. Download file `RXAWAR.ZIP`.



Visual programming is currently a hot issue in the desktop marketplace. This article demonstrates a visual programming tool in a sample-intensive manner, based upon a recent job assignment by one of the authors. By MARK A. BENGE, JOHN BOEZEMAN, and JAMES PIRKLE JR.

Visually Sampling DB2/2

As OS/2 gains acceptance in the marketplace, tools that promote application development for the operating system continue to arrive on the scene. One such tool uses a visual interface for program construction. Visual programming tools are designed to encourage rapid creation of applications. Our goal is to provide a sample-intensive article that highlights the strengths of visual programming.

We based part of the article on a programming assignment that James Pirkle recently completed for his client, R.J. Reynolds Inc. We demonstrate how easy it can be to generate an application that will query DB2/2 or, more specifically, access the sample database that ships with DB2/2. We will use Hockware's VisPro/REXX v. 1.1 to generate our DB2/2 sampling application. Additionally, we will show you how to use one of the new DB2/2 support features in the latest version, VisPro/REXX v. 2.0 Gold Edition.

BACKGROUND

VisPro/REXX is tightly integrated with the Workplace Shell. If you're familiar with the Workplace Shell, the development environment will be highly intuitive. Direct manipulation (**Drag&drop**) is used extensively in the development of applications. Additionally, as the name

implies, the product incorporates the OS/2 REXX programming facility. However, as you will see in our sample application, you do not have to be a REXX guru to use the product. Table 1 provides definitions of the terms that we use throughout the article.

SAMPLE APPLICATION

Our sample application, called DB2/2 Sampler, illustrates the use of various OS/2 GUI controls: the notebook, listbox, entry field, push and radio buttons, and value sets. You can obtain the source code for DB2/2 Sampler from the electronic sources listed in the references box on page 46. We will cover the various steps we followed to create DB2/2 Sampler using VisPro/REXX, so you can reference them.

CREATING A PROJECT

Our first task is to create a new project and forms for all of the DB2/2 Sampler windows. The project will encapsulate all the forms required by the sample application. Figure 1 contains these steps.

INITIALIZING AND ADDING NOTEBOOK PAGES

Notebook pages in Vispro/REXX are really just another form that you add to the form you have designated as a



Event	Occurrence that prompts behavior
Behavior	Function that a form or object performs
Object	A graphical representation of a control or controls
Project	A logical unit of operation such as the sample application
Form	Environment in which the application is designed
Canvas	Area in Layout view in which you add objects to the form

Table 1. Terms used throughout the article

- From the VisPro/REXX PROJECTS folder, drag and drop the Projectfolder template, which will cause a new project to be created.
- Direct Edit the new project folder and change the name to Demo. Open the Direct Edit window using a combination of the <Alt> key and mouse button 1. Close the window by clicking outside of the window.
- Open the Demo folder by double clicking on it with mouse button 1.
- Create the notebook, the two notebook pages, and the DB2/2 Query form.
 - i) Drag&drop the Form template within the Demo folder. This will create a new form. Repeat this three more times.
 - ii) Change the names of the new forms to Notebook, Colors, Fonts, and ShowDept.

Figure 1. Creating a new project

- Open the Fonts form, which will start VisPro/REXX.
- Choose the Open -> Settings menu item from the context menu on the canvas (click with mouse button 2 on the canvas to bring up the context menu).
- Select the Notebook page radio button and close the settings dialogue.
- Drag&drop a radio button object to the canvas.
- Change the text of the radio button to Courier 12pt. Repeat this step for the second Helv 18pt radio button.
- Double click on the top radio button. This will bring up the settings notebook for the radio button.
- Select group and close the dialogue window. Both radio buttons will be added to the same group if the group style is added to only one of them. As a result, only one of the radio buttons can be selected at a time.
- Select the Event tree view menu item from the View menu pulldown.
- From the context menu for the Courier radio button, select the When -> Clicked/selected menu item.
- Open the context menu on the edit region (window on the right). From the context menu, select Add -> Window management -> Notify source window, as illustrated in Figure 3. This will generate the following line:
CALL VpNotify window, 'message'
- Change message to FONT 12.Courier. This call will cause a text message to be sent to the parent of this window, which will be the notebook.
- Repeat the prior two steps for the FONT 18.Helv radio button.
- Save the Event tree view.
- From the Form menu pulldown, select Create link, which copies the REXX command for loading this page into the clipboard.
- Close the Fonts form.

Figure 2. Initializing Fonts page

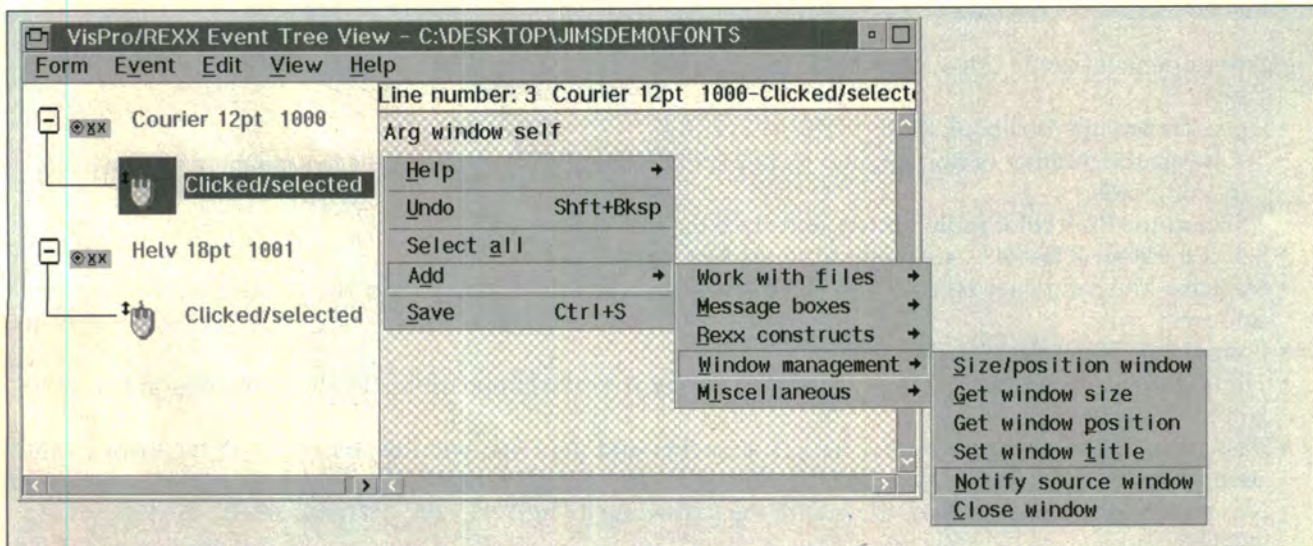


Figure 3. Source window notification step

- Open the Notebook form.
- Drag&drop a notebook to the canvas.
- Select the When -> Opened... submenu from the Form menu.
- From Edit menu on the Event window select Paste, which will put the REXX command that was copied into the clipboard, as shown in Figure 2, in the Event window. This statement will open the Font form when we test DB2/2 Sampler.
- Drag&drop the notebook from the canvas to the Event window. This will bring up the Create Link dialogue.
- Select the Add notebook page item, and change the word title to Fonts. This statement will add the Fonts form to the notebook as a page.
- Save and close the Event window and the Notebook form.

Figure 4. Adding Fonts page

notebook. We will show you how to initialize and add a page to a notebook in this section.

The Fonts notebook form is used to represent a notebook page that will allow selection of two different fonts in DB2/2 Sampler. Radio button controls will be placed on the page to allow selection. Figure 2 lists the steps required to initialize the Fonts page.

Once the Fonts page is initialized, it is ready to be added to the Notebook form. Figure 4 lists the steps required to add the Fonts page.

The Colors notebook form is used to represent a notebook page that will allow selection of eight

different background colors in DB2/2 Sampler. A value set control will be placed on the page to allow selection. Figure 5 lists the steps required to initialize the Colors page.

Once the Colors page is initialized, it is ready to be added to the Notebook form. Figure 6 lists the steps required to add the Colors page.

DISPLAYING DATA FROM SAMPLE DB2/2 DATABASE

The ShowDept form is used to display data in the sample DB2/2 database that ships with DB2/2. Text fields are created on the form to display

data. Figure 7 lists the steps required to initialize the form.

THE MAIN FORM

The Main form is the form initially displayed when a VisPro/REXX application is invoked. Besides initializing the form, REXX code must be added in order to register the SQL dynamic link libraries (DLLs) and start using the sample DB2/2 database.

Due to space constraints and the lengthy steps required to initialize the Main form, we provide this information in a README file that is part of the source code file that you can download. Specific



- After opening up the Colors form and changing it to a notebook page, drag&drop a valueset object to the canvas.
- Open the Settings notebook for the valueset control.
 - i) Change the number of horizontal cells to 2 and the number of vertical cells to 4, so there will be a total of eight cells.
 - ii) Select the RGB color radio button, and close the Settings view.
- Select the When -> Opened... submenu from the Form menu.
- Drag&drop the value set object from the canvas to the Event window and the Create Link listbox will appear.
- From the listbox, select Set cell value.
- Highlight the generated REXX statement and copy it to the clipboard by selecting the Copy menu item from the Edit menu.
- Paste from the clipboard seven of these statements into the Event window, by selecting the Paste menu item from the Edit menu.
- Replace the respective variable value with the following: 0, 16777215, 255, 16711680, 65280, 16776960, 65535, 16711935.
- Next replace the row, column variables with 1,1 2,1 3,1 4,1 1,2 2,2 3,2 4,2.
- Save and close the Event window.
- Open the Event tree view.
- Select the When -> Clicked/selected option in the context menu.
- Drag&drop the valueset over to the edit region.
- Select Get item value from the listbox.
- Open the context menu on the edit region and select Add -> Window management -> Notify source window. This will generate the following line:

```
CALL VpNotify window, 'message'
```

Change the message to COLOR value. This call will cause a text message to be sent to the parent of this window, which will be the notebook.
- Save the Event Tree View.
- Select Create Link from the Form menu.
- Close the Colors form.

Figure 5. Initializing Colors page

- Open the Notebook form.
- Select the When -> Opened menu item.
- From Edit menu on the Event window select Paste, which will put the REXX command that was copied into the clipboard, as shown in Figure 4, in the Event window. This statement will open the Colors form.
- Drag&drop the notebook from the canvas to the Event window.
- Select the Add notebook page item and change the word title to Colors. This statement will add the Colors form to the notebook as a page.
- Save and close the Event window.
- Select the When -> Secondary notify... menu item from the Form menu.
- Add the word Parse before the word Arg in the Event window.
- Open the context menu and select Add -> Window management -> Notify source window and change message in the resultant REXX statement to message.
- Save and close the Event window.
- Save and close the Notebook form.

Figure 6. Adding Colors page



- Open the ShowDept form.
- Drag&drop a push button to the form and edit the text to Ok.
- Drag&drop two text fields to the form, one for a label and the other to display the department information. Open the settings notebook for each and set the Symbol field to TEXT_LBL for the first and TEXT_DEPT for the other.
- Create a When Opened event for the form.
 - i) Select When -> Opened menu item from the Form menu.
 - ii) Open the context menu for the Event window and select Add -> Window management -> Set window title. Change the value variable to Department.
 - iii) Drag&drop the TEXT_LBL text field from the layout view to the Event window. When the Create Link listbox appears, select the Set item value item. Change the value variable to:
 'Department Mgr Division Location'.
 - iv) Append the following lines of REXX code:

```
errmsg = SELDEPT(window topic 'TEXT_DEPT')
if errmsg \= '' then
    signal SHOW_ERROR
RETURN 0
SHOW_ERROR:
```
 - v) Open the context menu for the Event window and select the Add -> Message boxes -> Plain menu item. Change title to Demo Error, and change message to errmsg.
 - vi) Open the context menu for the Event window and select the Add -> Window management -> Close window menu item.
 - vii) Save and close the Event window.
- Open the Event Tree View.
 - i) Open the context menu on the Ok pushbutton, and select the When -> Clicked/selected menu item.
 - ii) Open the context menu on the edit region and select the Add -> Window management -> Close window menu item.
 - iii) Save and close the Event Tree View.
- Save and close the ShowDept form.

Figure 7. Initializing ShowDept form

information on the file is in the references box on page 46.

SUBROUTINES PROCEDURES (SUBPROCS)

SubProcs are segments of REXX code that can be shared by events within a VisPro/REXX application.

Additionally, they are created with the editor of your choice, the only stipulation being that they reside in the XXX\SubProcs subdirectory, where XXX is the name of your project.

Three SubProcs are defined for the DB2/2 Sampler application, and they provide the following services:

- SELSTAFF formats and adds staff information to the listbox on the

Main form.

- SELDEPT formats and displays selected department information.
- OPENDB registers SQL DLLs and connects to the DB2/2 database.

The source for the three SubProcs can be obtained from the electronic sources listed in the references box on page 46.

TESTING DB2/2 SAMPLER

Finally, we are ready to test DB2/2 Sampler. To test the application, select the Test menu item from the Form menu on the Main form.

If things didn't turn out the way you planned them, make modifications or refer to the sample source code.

In Figure 8, we illustrate a test session. In this illustration, the Open Settings notebook has been opened.

BUILDING AN EXECUTABLE PROGRAM

If everything works as you desire, you can build an executable program by selecting the Build menu item from the Form menu on the Main form.

NEW DB2/2 SUPPORT IN VISPRO/REXX V. 2.0

As you can see, a great deal of our time is spent adding REXX code to perform the queries to the sample DB2/2 database. If this code were generated for us,

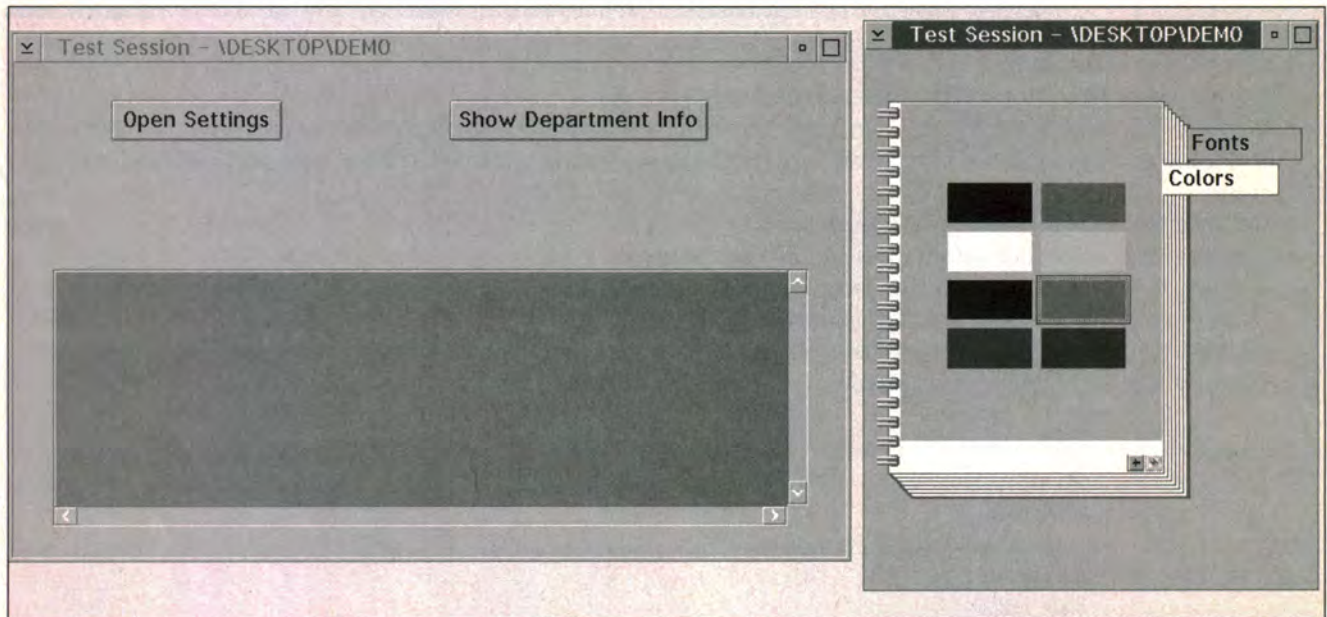


Figure 8. Testing DB2/2 Sampler

- Create a diagram by dragging one off of the VisPro/REXX database diagram template.
- Open the diagram.
- From the DataBase menu, select the Reverse Engineer menu item.
- Select the SAMPLE database from the listbox, click on Find Qualifiers, select USERID from the listbox, and click on OK. The sample database window appears as shown in Figure 10. Observe how the diagrammer has identified the referential constraint relationship between the STAFF and ORG tables.
- Highlight the ORG table object. From its context menu, select Selected -> Open Settings. When the Open Settings notebook appears, select the Attributes tab and the notebook page shown in Figure 11 will appear.

Figure 9. Reverse engineer

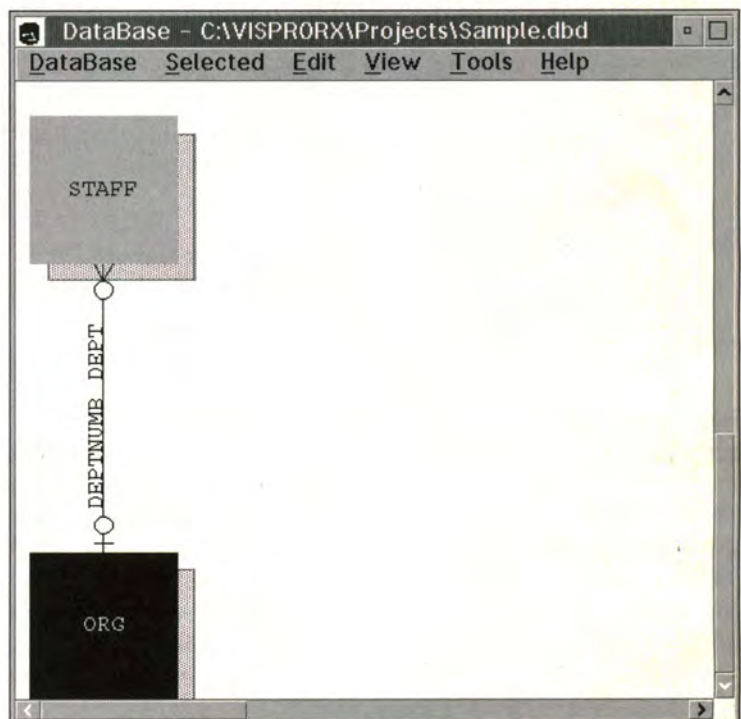


Figure 10. Referential constraint relationship

not only would we immediately become more productive, we would reduce the size of the maintainable code base, which is a blessing to those whose task is program maintenance.

The database support fea-

tures that are in VisPro/REXX v. 2.0 Gold Edition are a logical extension to the v. 1.1 product. If you're comfortable with the work we've done here, the following information will be easy for you to digest.

PREPARING THE SAMPLE DATABASE

To generate a form to access a DB2/2 table, the reverse engineering tool requires the table to have a primary key. To add primary keys to the ORG and STAFF tables, you can enter the following commands at

Name	Type	Key	Null	Text
DEPTNAME	VCHAR N	Y	Y	Y
DEPTNUMB	SINT Y	N	N/A	N/A
DIVISION	VCHAR N	Y	Y	Y
LOCATION	VCHAR N	Y	Y	Y
MANAGER	SINT N	Y	N/A	N/A

Figure 11. ORG table attributes

DB2/2's command line interface (we also added a foreign key to the STAFF table to illustrate VisPro/REXX's Database Diagrammer):

1. STARTDBM
2. LOGON userid /P:password
3. DBM CONNECT TO SAMPLE
4. DBM ALTER TABLE ORG PRIMARY KEY (DEPTNUMB)
5. DBM ALTER TABLE STAFF PRIMARY KEY (ID) FOREIGN KEY FOR_DEPT (DEPT) REFERENCES ORG ON DELETE RESTRICT

REVERSE ENGINEERING THE SAMPLE DATABASE

Since the sample database already exists, we can take advantage of one of the new features that will allow us to reverse engineer the

sample database. Reverse engineering will display the structure of the database in a graphical format. The steps required to reverse engineer are listed in Figure 9.

THE MAIN FORM REVISITED

In the previous sections that dealt with the Main form and SubProcs, many steps were taken to ensure that all the REXX code was in place to query the sample database. The steps illustrated in Figure 12 replace many of the steps required to initialize the Main form and all of the REXX code entered for the three SubProcs, which is not required in v. 2.0. As you will see, the time and effort required to create applica-

tions by reverse engineering existing databases and Drag&drop is considerably less than what is required to build similar applications by traditional programming techniques.

The steps in the Main form that are not replaced involve adding the Open Settings push button and creating a link to the Notebook form, so it will be opened when a user clicks on Open Settings. Also, the links required to change the Font and Color will need to be implemented based upon the new DB2/2 support.

CONCLUSION

Visual programming encourages rapid prototyping and application development. When compared to a conventional command prompt development environment, the time savings can be measured in hours or even days. However, the pace does slow when you are required to enter code to add support that is not present in the visual programming tool. As the state of these tools advance, so does their functionality.

In the section, *New DB2/2 Support in VisPro/REXX v. 2.0*, we looked at a recent enhancement to the visual programming environment. VisPro/REXX saved not only time but also wear and tear on the keyboard. Unfortunately, we could cover only the reverse engineering portion of the Visual Database Designer enhancement, which can also be used to design DB2/2 databases from scratch.

- Open the Main form.
- Drag&drop the ORG table object onto the Main form. The Create Database Link dialogue will appear, as shown in Figure 13. Use all of the columns and action buttons, which are already selected by default, by clicking on the OK button.
- Save the Main form.
- Test the demo application by selecting the Test menu item from the Form menu, and the test dialogue as shown in Figure 14, will appear. Note the support that allows you to Add, Delete, Change, and Search rows. In the DB2/2 Sampler application, additional code is required to perform these tasks, whereas a simple checkbox selection generates the code for us in v. 2.0 release.

Figure 12. Revisiting the Main form

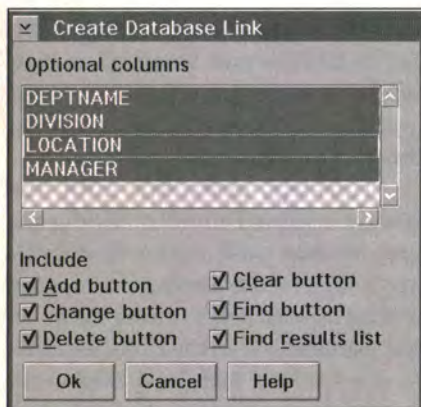


Figure 13. Create database link dialogue

Mark A. Bengé, IBM Programming System Laboratory, Cary, N.C., is a staff programmer who joined IBM in 1989 and has worked on various CUA '91 controls for OS/2 2.x and CUA Controls Library/2. He works in IBM user interface class library development, where he was involved in the implementation of C++ classes for direct manipulation for the C Set++ 2.1 product. Bengé has a B.S. in computer science from Western Carolina University.

John Boezeman, IBM Software Solutions Laboratory, Cary, N.C., is a senior associate programmer who joined IBM in 1990 and has since worked on various C++ projects. He now works in IBM class library user interface development, where he is involved in the implementation of C++ classes for multimedia. Boezeman received a B.S. in computer science from Indiana University.

James Pirkle, Jr., J. Pirkle Inc., Pfafftown, N.C., is an independent consultant with over 10 years of experience writing software for real-time systems. His most recent work includes development of a distributed inventory system using FM hand-held terminals for data acquisition and distributed databases using DB2/2 and MVS host-based databases.

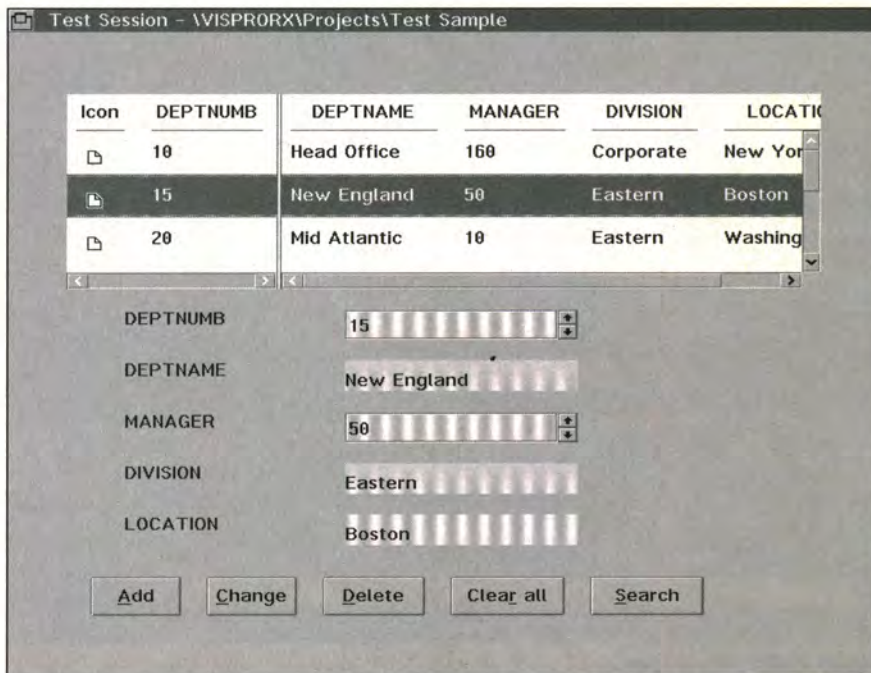


Figure 14. Testing DB2/2 Sampler using new DB2/2 support

REFERENCES

VisPro/REXX User Manual

Database 2 OS/2 Guide (IBM Doc. S62G-3663-00)

Database 2 OS/2 SQL Reference (IBM Doc. S62G-3665-00) Database 2

OS/2 Programming Reference (IBM Doc. S62G-3666-00)

Database 2 OS/2 Programming Guide (IBM Doc. S62G-3667-00)

Procedures Language 2/ REXX Reference (IBM Doc. S10G-6268-00)

The source code can be downloaded from several electronic sources:

- In the U.S., call the IBM PCC BBS at (919) 517-0001. The source code for this article is in File Area 11 under the name VPRDB2.EXE.
- On CompuServe, the source code is in LIB13 of the OS2DF2 forum, under the name VPRDB2.EXE.
- If you're an IBM TALKLINK customer, the source can be downloaded from the OS2BBS.
- If you have a VM account on an IBM node, get the source code with the command REQUEST VPRDB2 PACKAGE FROM BANZAI AT CARVM3.



You can easily write additional functions to extend the function base available to the REXX programmer. This article provides an explanation and source code for a simple set of external functions written in C.

By ANDREI MALACINSKI and PATRICK MUELLER

Extending REXX with External Functions

The REXX language, provided as part of OS/2 2.0 and 2.1, is an easy-to-learn and powerful interpreted language. Besides offering a number of built-in functions, it allows users to add functions to the language by writing external functions in C. These external functions can be written to enhance the function set provided or to become small layers of code on top of a set of existing C functions, making a C function package available in REXX.

This article provides the source for a simple set of external functions written in C. The article notes differences between the C programming environment and the REXX programming environment and provides hints and pitfalls to avoid when writing external functions.

EXTERNAL FUNCTIONS

REXX contains a number of useful built-in functions. REXXUTIL.DLL, an external function package provided with OS/2, contains additional OS/2-related functions. Here, we describe how you can create external functions packaged in dynamic link libraries (DLLs), such as REXXUTIL.DLL.

The external functions that make up the DLL are written in C with a special interface that makes them callable from REXX. Each REXX function you provide needs a C function that implements the REXX function. More than one REXX function can use the same C function.

In C, a function prototype can have many formats: multiple parameters, call by reference, and call by value. C functions that implement REXX external functions must have the same prototype.

The prototype must correspond to the following:

```
ULONG APIENTRY FunctionName(
    PCHAR      pszName,
    ULONG      ulArgc,
    PRXSTRING  prxArgv,
    PSZ        pszQueueName,
    PRXSTRING  prxRet
)
```

The arguments are:

- **pszName.** This is the name of the REXX function invoked. When more than one REXX function is implemented by the same C function, this parameter is used to determine which REXX function was actually invoked.
- **ulArgc.** This is the number of parameters passed to the REXX function. It is similar to the **argc** parameter to the C **main()** function.
- **prxArgv.** This is an array of string parameters. It is similar to the **argv** parameter to the C **main()** function. However, instead of an array of **char ***, **prxArgv** is an array of **RXSTRING** structures. Each **RXSTRING** contains a parameter and its length.
- **pszQueueName.** This is the name of the currently defined REXX external data queue.
- **prxRet.** This is a pointer to an **RXSTRING** structure that the C function fills to return a value from the function to the REXX program.

THE RXSTRING STRUCTURE

In the REXX language, the only data type is the string. Parameters passed to external functions are strings, and the value returned from an external

function is a string. REXX strings are represented in C with the `RXSTRING` structure. It is defined as:

```
typedef struct
{
    ULONG strlength;
    PCH strptr;
} RXSTRING;
```

In C, strings always have a hex 00 character that indicates the end of the string. In REXX, all characters are valid in a string—even the hex 00 character. Therefore, the length of the string needs to be encoded outside of the string data. In the `RXSTRING` structure, the `strlength` field contains the length. The `strptr` field is a pointer to the string, like a `char *` variable in C.

When REXX passes arguments to external functions, it adds an extra hex 00 to the end of each argument. This makes it easy to use the `strptr` fields of the `RXSTRING` in C functions such as `strtod()` and `strtok()`. This feature was added in OS/2 2.0.

The REXX language allows empty (zero length) strings. The C equivalent is a `char *` variable that points to a hex 00 character. When a zero-length string is passed as an argument to a REXX external function, the `strlength` field is set to zero, but the `strptr` field is set to a nonzero value. It is also possible to omit parameters from a function call, as in:

```
FunctionName("a",,"b",,"b")
```

In this example, the second and fourth parameters are omitted. These parameters are passed to REXX, with both the `strptr` and `strlength` fields set to zero.

The C language include file from the IBM Developers Toolkit for OS/2 2.0, `REXXSAA.H`, contains a number of useful macros for implementing external functions. One such macro is `RXVALIDSTRING()`. In the sample code contained in Figure 1, this macro is used when checking parameters. This particular macro returns 1 if the `strptr` and `strlength` fields of an `RXSTRING` structure are both nonzero; otherwise 0 is returned.

```
=====
rxprior.mak
=====

#-----
# Makefile for rxprior.dll
#-----

COPTS = /C+ /W3 /Q+ /Gd- /Ge- /Gm+ /Ti+ /O-
LOPTS = /ST:20000 /NOLOGO /DEBUG

OBS = rxprior.obj
LIBS = os2386 rexx

.c.obj:
    icc $(COPTS) /Fo$@ $<

rxprior.dll : $(OBS)
    link386 $(LOPTS) $(OBS),$@,nul,$(LIBS),rxprior.def;

=====
rxprior.def
=====

LIBRARY RXPRIOR INITINSTANCE TERMINSTANCE
DATA MULTIPLE NONSHARED
EXPORTS
    SysSetPriority
    SysGetPriority

=====
rxprior.cmd
=====

/*-----
 * rxprior.cmd : test the external functions in rxprior.dll
 *-----*/

"@echo off"

/*-----
 * load functions
 *-----*/
if RxFuncQuery("SysSetPriority") then
do
    rc = RxFuncAdd("SysSetPriority","RxPrior","SysSetPriority")
    rc = RxFuncAdd("SysGetPriority","RxPrior","SysGetPriority")
end

/*-----
 * print the current priority
 *-----*/
say "current priority:" SysGetPriority()

/*-----
```

Figure 1. External function sample (continued on page 49)

PROCESSING THE EXTERNAL FUNCTION



The processing that an external function needs to perform can be broken into three steps:

1. Process the parameters passed to the function.
2. Perform whatever processing is required, based on the parameters.
3. Set the return value.

Processing parameters. Because REXX cannot prototype functions, the user can call external functions with any number of parameters. It's up to you to determine whether the parameters that are passed in are correct.

In the Figure 1, the C function that implements SysSetPriority() function uses the number of parameters that are passed in and the values from the REXX parameters to perform basic checks and provide default values. The first check,

```
if ((u1Argc < 1) || (u1Argc > 3))
    return 40;
```

causes the function to return 40 if no parameters or more than three parameters are passed to the function. If 40 is returned from an external function, it causes the REXX error, *Incorrect call to routine*, to be generated. This error is fatal to REXX programs; generate it only if absolutely required. The rest of the parameter-checking code in the SysSetPriority() function provides default values for missing or incorrect parameters that are passed in.

Recall that all parameters are passed in as strings. Therefore, the external function must convert the received arguments from string format to the appropriate data type. In the Figure 1, after checking the parameters, the SysSetPriority() function converts the strings passed in (or defaults) to the types of values it needs. It then calls the C function, *DosSetPriority()*, to do the real work.

Performing function processing. Once the parameters have been processed and converted into C values, you need to do whatever processing the external function is supposed to do. Generally, you can do whatever you want to do, within the

```
* change the priority a few times, and print info each time
*-----*/

call Test "-1", "NOCHANGE"      , "PROCESS"
call Test "-1", "REGULAR"       , "THREAD"
call Test "-1", "TIMECRITICAL"  , ""
call Test "-1", "FOREGROUNDSEVER", "GARBAGE"

call Test "0"
call Test "0", "NOCHANGE"
call Test "1", "NOCHANGE"
call Test "2", "NOCHANGE"
call Test "3", "NOCHANGE"

call Test "0", "REGULAR"

exit

/*-----*
* run one test
*-----*/
Test: procedure expose pid
    delta = arg(1)
    class = arg(2)
    scope = arg(3)

/*-----*
* call with different # of parameters to test defaulting
*-----*/
if (arg() = 1) then rc = SysSetPriority(delta)
else if (arg() = 2) then rc = SysSetPriority(delta,class)
else if (arg() = 3) then rc = SysSetPriority(delta,class,scope)

if (rc <> 0) then
    say "rc =" rc "from SysSetPriority("delta","class","scope")"

/*-----*
* get the priority
*-----*/
priority = SysGetPriority()

say "priority =" priority,
    "from SysSetPriority("delta","class","scope")"

return

=====
rxprior.c
=====

/*-----*
* rxprior.c : external rexx function to set priority
```

Figure 1. External function sample (continued on page 50)

capabilities of C. There are a few things to look out for:

- Be aware of what the requirements are for the functions that you are calling. For instance, if you intend to make calls to Presentation Manager functions, the current process will probably need to run in a PM session.
- Be careful of non-reentrant processing. Although there are few programs today that allow REXX macros to run in multiple threads within one process, this capability does exist. The same concerns for multithreaded programming in C are valid for external functions for REXX written in C.

Setting return value. To return a REXX value from the function, you must set fields in the `prxRet` structure, which is passed to the function. If the value returned is numeric, you must convert the number from the format C uses to a string, using `sprintf()`, for instance. The `strptr` of this particular `RXSTRING` structure points to a 256-character-long buffer. If the string returned from the function fits in the buffer, write the string to the buffer and set the `strlength` field to the length of the string. If the string is longer than 256 characters, allocate a buffer for it, using `DosAllocMem()`.

Note that you cannot use `malloc()` to allocate the buffer. Set the `strptr` field to the address returned by `DosAllocMem()`. REXX will call `DosFreeMem()` to free this memory after it has made a copy of it.

The external function written in C must return an unsigned long value. Remember that returning 40 from an external function causes a REXX error to occur in the program running the function; any nonzero return code from an external function will cause a fatal REXX error to occur. Make sure you return 0 from your external function, unless you specifically want to cause a fatal error.

REGISTERING THE EXTERNAL FUNCTIONS IN THE DLL

Before calling the external functions in the DLL from a REXX program, the REXX program must register them.

```

/*-----*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define INCL_BASE
#include <os2.h>

#define INCL_REXXSAA
#include <rexxsaa.h>

/*-----
 * prototypes for rexx functions, to make sure we declared everything
 * correctly
 *-----*/

RexxFunctionHandler SysSetPriority;
RexxFunctionHandler SysGetPriority;

#pragma linkage(SysSetPriority, system)
#pragma linkage(SysGetPriority, system)

/*-----
 * this function is called when user invokes SysGetPriority() in REXX
 *
 * expects no parameters
 *-----*/
ULONG APIENTRY SysGetPriority(
    PCHAR    pszName,
    ULONG    ulArgc,
    PRXSTRING prxArgv,
    PSZ      pszQueueName,
    PRXSTRING prxRet
)
{
    PPIB  ppib;
    PTIB  ptib;
    APIRET rc;

    /*-----
     * call DosGetInfoBlocks()
     *-----*/
    rc = DosGetInfoBlocks(&ptib,&ppib);

    /*-----
     * put priority in return code
     *-----*/
    sprintf(prxRet->strptr,"%08.8lx",ptib->tib_ptib2->tib2_ulpri);
    prxRet->strlength = strlen(prxRet->strptr);

    return 0;
}

```

Figure 1. External function sample (continued on page 51)


```

}

/*-----
* this function is called when user invokes SysSetPriority() in REXX
*
* expects the following parameters
* 1 - number between -31 and 31, the delta of the priority to set
* 2 - optional class: "NOCHANGE", "IDLETIME", "REGULAR",
*                   "TIMECRITICAL", "FOREGROUNDSEVER"
* 3 - option scope: "PROCESS", "PROCESSTREE", "THREAD"
*-----*/
ULONG APIENTRY SysSetPriority(
    PCHAR    pszName,
    ULONG    ulArgc,
    PRXSTRING prxArgv,
    PSZ      pszQueueName,
    PRXSTRING prxRet
)
{
    APIRET    rc;
    PSZ      pszDelta;
    PSZ      pszClass;
    PSZ      pszScope;
    ULONG    ulScope;
    ULONG    ulClass;
    LONG     lDelta;

    /*-----
    * make sure there are 1, 2 or 3 parameters
    *-----*/
    if ((ulArgc < 1) || (ulArgc > 3))
        return 40;

    /*-----
    * apply default parameters
    *-----*/
    if (RXVALIDSTRING(prxArgv^0))
        pszDelta = prxArgv^0 .strptr;
    else
        pszDelta = "0";

    if ((ulArgc >= 2) && RXVALIDSTRING(prxArgv^1))
        pszClass = prxArgv^1 .strptr;
    else
        pszClass = "NOCHANGE";

    if ((ulArgc >= 3) && RXVALIDSTRING(prxArgv^2))
        pszScope = prxArgv^2 .strptr;
    else
        pszScope = "PROCESS";

```

Figure 1. External function sample (continued on page 52)

To do this, use the `RxFuncAdd()` function. The `RxPrior.cmd` file provides an example of registering the two external functions given in `RxPrior.dll`.

Registering a function associates the function name used in REXX programs with a DLL and an entry point in the DLL (the name of the C function). You can use the same entry point for more than one REXX function. This is useful if several external functions perform similar processing. By examining its first parameter, the external function can determine which function was actually invoked.

Once an external function is registered, it is available to any REXX program in the system. Thus, the person using the external functions only needs to register them once, for example, in `STARTUP.CMD`. A REXX program can check if an external function has already been registered with the `RxFuncQuery()` function.

External functions can also be registered in C. This can be useful with a DLL that has a large number of functions. Instead of requiring the user of the external-function package to call `RxFuncAdd()` for each function, you can provide one external function that loads all the other functions. External functions can be registered from C with the function `RexxRegisterFunctionDll()`.

The `REXXUTIL.DLL` provided with OS/2 uses this technique to register its functions. First, you register the function `SysLoadFuncs` in REXX with the `RxFuncAdd()` function, then you call it. This function calls `RexxRegisterFunctionDll()` on the rest of the functions. `REXXUTIL.DLL` also provides a function, `SysDropFuncs`, to unregister the functions. Unregistering a function is a system-wide operation, much like registration. After unregistering a function, the function will be unavailable to REXX programs running on the system, until it is registered again.

COMPILING AND LINKING THE EXTERNAL FUNCTIONS

When writing external REXX functions in C, include the following lines for each external function you write:




```
RexxFunctionHandler FunctionName;
#pragma linkage(FunctionName, system)
```

The first statement prototypes the function. If you use the incorrect argument types or forget an argument, this prototype will cause a compiler error to be generated. An external function that does not have the correct prototype can cause the external function to behave incorrectly or cause an access violation.

The `#pragma` statement ensures that the function has the system linkage (as opposed to the `optlink` linkage) for C Set/2.

A make file is provided with the sample in Figure 1. The options used will compile and link the source with C Set/2 compiler and the `Link386` program.

A `.def` file is also included. Each of the external functions in C needs to be exported from the DLL.

The DLL must be linked with `REXX.LIB`, in addition to the standard C libraries and any additional libraries that are needed.

DEBUGGING THE EXTERNAL FUNCTIONS

External functions contained in a DLL can be debugged with the IPMD debugger. Invoke IPMD with the command:

```
IPMD cmd /c myprog
```

where `myprog.cmd` is a REXX `.CMD` file containing calls to the external functions. Once IPMD starts, set a load breakpoint on your DLL and have IPMD start the process. Your DLL will load when the first function is called in it. At this point, set breakpoints, single step, and so on, within the debugger.

If you need to recompile your DLL after using it, the link step may fail with an error indicating that the run file (the DLL) cannot be opened. This is because when a REXX program is run from a `CMD.EXE` command prompt, it is run in the current OS/2 process. Specifically, your current `CMD.EXE` session has the DLL loaded. In fact, all of the currently open `CMD.EXE` sessions might have the DLL loaded. OS/2 does not allow DLLs

```
/*-----
 * convert delta to a number
 *-----*/
lDelta = strtol(pszDelta,NULL,10);

/*-----
 * convert class to a pre-defined constant
 *-----*/
if (!strcmp(pszClass, "NOCHANGE"))
    ulClass = PRTYC_NOCHANGE;
else if (!strcmp(pszClass, "IDLETIME"))
    ulClass = PRTYC_IDLETIME;
else if (!strcmp(pszClass, "REGULAR"))
    ulClass = PRTYC_REGULAR;
else if (!strcmp(pszClass, "TIMECRITICAL"))
    ulClass = PRTYC_TIMECRITICAL;
else if (!strcmp(pszClass, "FOREGROUNDSEVER"))
    ulClass = PRTYC_FOREGROUNDSEVER;

else
    ulClass = PRTYC_NOCHANGE;

/*-----
 * convert scope to a pre-defined constant
 *-----*/
if (!strcmp(pszScope, "PROCESS"))
    ulScope = PRTYS_PROCESS;

else if (!strcmp(pszScope, "PROCESSTREE"))
    ulScope = PRTYS_PROCESSTREE;

else if (!strcmp(pszScope, "THREAD"))
    ulScope = PRTYS_THREAD;

else
    ulScope = PRTYS_PROCESS;

/*-----
 * call the DosSetPriority() function
 *-----*/
rc = DosSetPriority(ulScope,ulClass,lDelta,0);

/*-----
 * set return value
 *-----*/
sprintf(prxRet->strptr,"%ld",rc);
prxRet->strlength = strlen(prxRet->strptr);

return 0;
}
```

Figure 1. External function sample (continued from page 51)

that are currently loaded to be erased or written to. To remove the DLL, you need to unregister each of the functions. External functions can be unregistered with the `RxFuncDrop()` function in REXX or the `RexxDeregisterFunction()` function in C. After unregistering the functions, you must exit all currently open CMD.EXE sessions. OS/2 will then drop the DLL from the system. At this point, you can start new CMD.EXE sessions and erase or relink the DLL.

REXX MACROS IN THE CURRENT PROCESS

REXX macros have a significant difference from other OS/2 executable files: they do not start a new process. A REXX .CMD file, run from a CMD.EXE session, runs in the same process as the CMD.EXE program. OS/2 does not start a new process for the program or end it when the program terminates.

This is important, because OS/2 resources are not cleaned up when a REXX macro ends. For example, memory allocated with `malloc()` or `DosAllocMem()` from an external function is not freed when the REXX program that used the external function ends. If you need to use these functions to allocate memory from within an external function, free the memory before the external function returns or it will remain allocated within the CMD.EXE process. If you forget to free the memory, you won't be able to free it later because you will have lost the pointer value. The only way to free the memory will be to exit the current CMD.EXE process.

This also has implications for `DosExitList()` processing. If your external

function uses another DLL that depends on `DosExitList()` processing, this processing won't occur until after the CMD.EXE process has terminated.

You can get around this limitation by writing a C program that invokes REXX macros with the `RexxStart()` function. The program can register the external functions before starting the REXX macro with the `RexxRegisterFunctionDll()` or `RexxRegisterFunctionExe()` functions. The first argument to the program should be used as a REXX macro name, and the remainder should be concatenated to form the parameter string to be passed to the REXX macro.

This C program can then be used with the `EXTPROC` facility of OS/2. In a .CMD file, make the first line:

```
EXTPROC myCprog
```

where `myCprog` is the name of the C program.

When OS/2 runs the .CMD file, it recognizes the `EXTPROC` statement and starts the `myCprog` program with the parameters specified on the command line for the .CMD file. The C program is then in control. A new process starts and will terminate when the program ends. Therefore, OS/2 resources, such as memory and file handles, will be cleaned up when the `myCprog` program ends.

You should unregister the REXX functions that were registered by the C program, before the C program exits.

EXTERNAL FUNCTION SAMPLE

The sample code implements two functions: `SysGetPriority()` and `SysSetPriority()`. These functions are used to query and set the priority of the current process.

SysGetPriority(). This function expects no parameters. It returns an eight-digit hex string that indicates the priority of the current process. The function returns the priority from a field in the infoBlocks structure returned by the `DosGetInfoBlocks()` function.

SysSetPriority(delta, class, scope). This function expects the delta parameter to be a number from -31 to 31; the class parameter to be one of the strings, `NOCHANGE`, `IDLETIME`, `REGULAR`, `TIME-CRITICAL`, or `FOREGROUNDSEVER`; and the scope parameter to be one of the strings, `PROCESS`, `PROCESSTREE`, or `THREAD`. The parameters correspond to the values expected by the `DosSetPriority()` function. The external function calls this function and returns the return code from that function. The delta parameter must be passed in, but the other two are optional. Missing or incorrect parameters cause default values to be used.

`RxPrior.cmd` tests the external functions by calling `SysSetPriority()` a number of times with different parameters. After each call to `SysSetPriority()`, `SysGetPriority()` is called to determine what the priority was set to. The resulting value is printed as output from the program.

CONCLUSION

The REXX language allows for expansion of its function base through external functions. Based on that design, and with the help of these tips and explanations, you are capable of extending the language to meet your needs. Go for it!

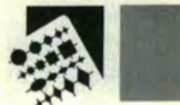
Andrei Malacinski, IBM Programming Systems, Cary N.C., is a member of the Distributed Application/2 programming team. He joined IBM in 1990, after receiving degrees in mathematics and computer science from Indiana University.

Patrick Mueller, IBM Programming Systems, Cary N.C., is a member of the Object Oriented Strategy Department. He joined IBM in 1985, after receiving degrees in mathematics and computer science from Purdue University. He can be reached at pmueller@vnet.ibm.com.

REFERENCES

OS/2 2.0 Technical Library, Procedures Language 2/REXX User's Guide, (IBM Doc. S10G-6269).

OS/2 2.0 Technical Library, Procedures Language 2/REXX Reference, (IBM Doc. S10G-6268).





Buyer's Guide

The staff of OS/2 Developer put together this special section to guide you through the various REXX products available for OS/2. The products described in this guide are based on surveys sent to vendors; they are provided as a service to you and are free to vendors. These listings do not represent an endorsement by OS/2 Developer. Although every effort has been made to ensure accuracy, we do not assume any responsibility for error or omission in this section. Please contact the companies directly for more information.

REXX Buyer's Guide

DSOFT DEVELOPMENT INC.

dbfREXX 1.0 is a DLL that extends the REXX language. It enables REXX programs to read and write dBASE files and indexes; supported file formats are DBF, DBT, and NDX. Price: \$59.00 through July 31, 1994, \$99.00 thereafter.

dSOFT Development Inc., 4710 Innsbruck Dr., Houston, Tex. 77066, Sales (405) 360-3045, Tech Support (713) 537-0318, fax (713) 537-0318.

GPF SYSTEMS

GpfRexx 1.0 combines WYSIWYG visual programming with REXX. The user points and clicks to create CUA '91 applications and can take advantage of multimedia, drag and drop, multitasking, SQL (DB 2/2), and advanced communications such as APPC, CPI-C, and EHLLAPI. The Gpf Interface Builder handles the logic for displaying help and for managing events, windows, controls, and their data. No royalties. Price: \$247.50.

Gpf Systems, 10 Falls Rd., Moodus, Conn. 06469, (800) 831-0017 or (203) 873-3300, fax (203) 873-3302.

HOCKWARE INC.

VisPro/REXX 2.03 takes the power of OS/2, Workplace Shell, and REXX language and harnesses them into an easy-to-use visual programming environment. VisPro/REXX provides a development environment in which royalty-free programs are quickly cre-

ated, tested, debugged, modified, and given to as many users as desired. Price: Bronze Edition \$99.00, Gold Edition \$299.00.

VisPro/REXX Data Entry Object Pack is an add-on, stand-alone package of five programming objects that allows programmers to add commonly used business objects to their VisPro/REXX application, such as formatted entry field, spreadsheet, split bar, clock, and calendar. It requires VisPro/REXX Bronze or Gold 2.x or higher. Price: \$119.00.

Hockware Inc., 315 N. Academy St., Ste. 100, Cary, N.C. 27513, (919) 380-0616, fax (919) 380-0757.

HSS INC.

Panel Display System 1.04 is a character user interface alternative to Presentation Manager. It adds interactive capabilities to REXX through the use of PDS panels defined via a panel definition language and displayed via the REXX PDS API. PDS handles user interaction with the panels, and the RXPDI application processes the results. Prices: Base \$10.00, REXX API \$5.00, HLL API \$15.00.

HSS Inc., 1635 Village Glen Dr., Raleigh, N.C. 27612, (919) 881-0654, fax (919) 380-0757.

MANSFIELD SOFTWARE GROUP INC.

KEDIT 5.0 is a text editor that uses REXX for macros and key definitions through either a



built-in subset of REXX, Personal REXX, or IBM's OS/2 REXX. It features XEDIT compatibility, multiple files and windows, redefinable keys, undo and redo, file locking, on-line help, macro debugger, and mouse support. Price: DOS version is \$180.00, 32-bit OS/2 and DOS version is \$210.00.

Mansfield Software Group Inc., P.O. Box 532, Storrs, Conn. 06268, (203) 429-8402, fax (203) 487-1185.

QUERCUS SYSTEMS

Personal REXX for OS/2 3.0 augments OS/2 REXX by including Quercus Systems' REXXLIS extensions for array handling, mathematical functions, interprocess communication, and general system services. It also provides global variables, additional utilities for VM/CMS and MVS/TSO compatibility, and external data queue support. Various performance improvements increase operating speed of OS/2. Price: \$175.00.

RexxComm 1.0 is a REXX function package that provides easy access to serial ports directly from REXX. High-level functions support sending and receiving data, matching multiple character strings, controlling a modem, and dialing a number. Intermediate-level functions help manage port configuration parameters, time-out intervals, and RS232 control signals. File transfer support includes Xmodem, Ymodem, Zmodem, Kermit, and Compuserve-B. Price: \$75.00.

REXXTERM 2.3 is an asynchronous communications program that provides support for user customization and script writing with REXX. It includes multiple dialing directories, built-in text editor, host mode, flexible keyboard configuration, VT102 and VT52 terminal emulation, Xmodem, Ymodem, Zmodem, Kermit, and Compuserve-B file transfer protocols. Price: \$100.00.

REXXLIB 1.0 is a collection of over 150 REXX extension functions in five areas: compound variable handling, interprocess communication, mathematical functions, system services, and text-mode user interfacing. Its functions include conversion of date formats, retrieval of tails of a compound variable, sorting and other array operations, regular expression searches, and reading or writing

collections of variables in disk files. Price: \$50.00.

Quercus Systems, P.O. Box 2157, Saratoga, Calif. 95070, (408) 867-7399, fax (408) 867-7489.

SOFTOUCH SYSTEMS INC.

Gammatech REXX Superset/2 Software Package

1.0 is an extension of REXX external functions. There are more than 300 functions in seven DLL'S to call from any REXX program. These program functions perform math calculations, initiate file and system operations, manipulate processes and semaphores, regulate the macrospace, and execute video I/O and issue network commands. Price: \$79.95.

SofTouch Systems, Inc., 1300 S. Meridian Ste. 600, Oklahoma City, Okla. 73108-1751, (405) 947-8080, fax (405) 632-6537.

STRATEGIC SOLUTIONS INTERNATIONAL CORP.

NetImpact PM/Workstation 1.0 is a Presentation Manager-based graphical user interface for network operations to communicate with NetImpact SNA/Agent. Displays real-time status of VTAM-managed resources and applications. It provides an interface for problem management and electronic messaging and REXX-based scripting facilities for VTAM commands and network management automation. It also measures and reports service level objectives for availability and reliability. Does not require NetView or Solve:Automation but can coexist with either. Extensive use of containers provides operators with individualized displays of network outage information via sorting and filters. Price dependent on configuration.

Service Point/32 v1.6 is an automation framework for OS/2- and NetView-based command and control over network element management systems. It uses SNA management services to send alert NMVTs and NetView operator messages and to receive and respond to NetView RUNCMDs. Scripting facilities are available from REXX, including GENALERT, RUNCMD, MSG, and VIEW processing. Interactive REXX scripts can run on OS/2 and in NetView



without modification (including 3270 view panel interface). Price: \$3,000.00.

Strategic Solutions International Corp., 1075 Tolland Turnpike, Manchester, Conn., 06040, (203) 649-1900, fax (203) 649-1230.

TRITUS INC.

Tritus SPF 1.2.7 is a 32-bit ISPF/PDF text editor for OS/2. It edits files up to 256MB and includes REXX edit Macros, mappable keyboard, unlimited undo/redo, EBCDIC support, Micro Focus Workbench integration, modifiable panels, cut/paste, text search, regular expressions, custom record delimiters, recordable keyboard macros, DOS support, and more than 660 pages of documentation. Price: \$195.00.

Tritus, Inc., 3300 Bee Caves Rd., Ste. 650, Austin, Tex. 78746, (800) 321-2100 or (512) 794-5800, fax (512) 794-3833.

VTS — DATENSYSME GRUBLT & CO.

REXXLAN/2 1.22 gives access to LAN Server API. The user can use more than 90 API calls from REXX programs including the new LS 3.0 Domain Controller Database API. REXxLan/2 tests LS API, prototypes network applications, and writes tools for use of LAN Requester. It requires OS/2 2.x and LS 1.3 or higher. Price: \$198.00 plus \$20.00 shipping.

VTS — Datensysteme Grublt & Co., P.O. Box 305583, 20317 Hamburg, Germany, +40-418124, fax +40-453873.

WATCOM

WATCOM VX REXX 2.0 is an environment for creating applications that exploit the graphical user interface capabilities

of OS/2 2.x and Presentation Manager. It combines a project management facility, visual designer, and an interactive source-level debugger to deliver an approachable and productive visual development environment. Price: \$199.00.

Watcom, 415 Phillip St., Waterloo, Ontario, Canada N2L 3X2, (519) 886-3700, fax (519) 747-4971.

BOOKS

VAN NOSTRAND REINHOLD

OS/2 2.1 Handbook: Basics, Applications, and Tips by **Halette German** is a guide on use for REXX interpreters and compilers in OS/2 and across other platforms. It includes a complete REXX tutorial, specific techniques for applications development, and reviews of third-party and shareware interfaces with OS/2 REXX.

Van Nostrand Reinhold, 115 5th Ave., New York, N.Y. 10003, (800) 842-842-3636 or (212) 254-3232, fax (212) 475-2548 or fax orders (606) 525-7778.

CFS NEVADA INC.

The REXX Reference Summary Handbook is 160 pages and details instructions and functions in SAA REXX distributed with OS/2. It describes functions included in the REXXLIB and RXWIN-DOW external function packages available from Quercus Systems. The handbook includes details for creating, maintaining, and manipulating Workplace Shell objects in REXX and has a 30-page cross-referenced index. The novice user and REXX veteran can easily locate needed functions and identify various subjects. Price: \$19.95 plus \$3.50 shipping and handling. Dis-

counts are available for quantity orders and to current owners of the first edition.

CFS Nevada, Inc., 953 E. Sahara Ave., Ste. 9B, Las Vegas, Nev. 89104-3012, (800) 739-9672 for orders or (702) 732-9616, fax (702) 732-3847.

MACMILLAN COMPUTER PUBLISHING

Teach Yourself REXX in 21 Days teaches basic programming concepts, the REXX language, and the use of REXX extensions supplied with OS/2. The reader becomes proficient with REXX and learns to program in the OS/2 environment. The book includes programming examples, summaries, and exercises.

MacMillan Computer Publishing, 201 West 103rd St., Indianapolis, Ind. 46290, (317) 581-3575, fax (317) 581-4675, for orders (800) 428-5331 or fax (800) 448-3804.

JOHN WILEY & SONS INC.

Application Development Using OS/2 REXX by Anthony Rudd provides a source of information necessary for developing applications using REXX in an OS/2 environment. Beginners learn steps required to develop REXX applications. Experts get a reference to aspects of REXX. Worked examples make complex concepts understandable. The book emphasizes programming interfaces.

Mastering OS/2 REXX has a step-by-step approach to help programmers learn what they can do with REXX, where REXX fits in with OS/2, and how to create and execute a REXX program. The book includes REXX syntax, some simple REXX verbs, and debugging.

John Wiley & Sons, Inc. 605 Third Ave., New York, NY 10158, (212) 850-6630, fax (212) 850-6799.



Conferences,
October 3 - 7, 1994
Exhibition,
October 4 - 6, 1994

Washington Convention Center
Washington D.C.

Come See the Software and Application Development Industries Come Alive!

"The best speakers, the best demonstrations, and the best exhibition!" *Terry McIntosh, Analyst - Chevron Corp.*

SD '94 East delivers development solutions across the spectrum of tools from C++ to Powerbuilder.

- Client/Server Tools
- Object-Oriented Tools
- Rapid Application Development Tools
- Database Design and Modeling Tools
- CASE Tools
- GUI Front Ends
- Languages
- Configuration Management Tools
- Testing and Debugging Tools
- Porting Tools
- Multimedia Tools

SD '94 East provides powerful, real-world, real-time education in 4 jam-packed conferences.

SD was the first and continues to be best at bringing you practical, unbiased guidance in over 200 classes across 17 tracks.



Join us at SD '94 East as the industry comes alive...

...on the show floor, in the conference sessions and in a series of special events with key industry players.

- 4 Plenary sessions include:
Mitchell Kertzman, CEO, Powersoft
Christine Comaford, *PC Week*
Vaughan Merlyn, Partner, Ernst & Young
- Hours of free technical sessions by Microsoft, Powersoft, Intersolv and others
- The C++ Programming Superbowl



Tools and Techniques for Software and Application Developers

Call for a Brochure!

1-800-441-8826
415-905-2784
Internet: sd94east@mfi.com
FAX: 415-905-2222



REXX...

branching out from the basics

REXXCOMM™

A REXX function package that provides easy access to serial ports directly from REXX. High-level functions support sending and receiving data, matching multiple character strings, controlling a modem, and dialing a number. Intermediate-level functions help manage port configuration parameters, time-out intervals, and RS232 control signals. File transfer support includes Xmodem, Ymodem, Zmodem, Kermit, and CompuServe-B.

Just released!

REXXLIB™

A collection of over 150 REXX extension functions in five principal areas: compound variable handling, interprocess communication, mathematical functions, system services, and text-mode user interfacing. Functions include conversion of date formats, retrieval of "tails" of a compound variable, sorting and other array operations, "regular expression" searches, and reading or writing collections of variables in disk files.

PERSONAL REXX®

Augments OS/2 REXX by including Quercus Systems' REXXLIB extensions for array handling, mathematical functions, interprocess communication, and general system services. In addition, it provides global variables, additional utilities for VM/CMS and MVS/TSO compatibility, and enhanced external data queue support. A variety of performance improvements make Personal REXX anywhere from 20% to 200% (or more) faster than the standard OS/2 REXX.

REXXTERM®

A full-featured asynchronous communications program that provides extensive support for user customization and script writing with REXX. Other major features include multiple dialing directories, built-in text editor, host mode, flexible keyboard configuration, VT102 and VT152 terminal emulation, and Xmodem, Ymodem, Zmodem, Kermit, and CompuServe-B file transfer protocols.

QUERCUS SYSTEMS

P.O. Box 2157
Saratoga, CA 95070

Phone: 408-867-7399

Fax: 408-867-7489

BBS: 408-867-7488

CompuServe: GO QUERCUS

Call 1-800-440-5944 for orders or information.

Mention this ad for free overnight shipping on
REXXCOMM, REXXLIB, Personal REXX for OS/2 or REXXTERM.

Offer expires September 30, 1994

(Destinations outside the continental U. S. receive free shipping via U. S. Airmail.)

Also available: *Programming in REXX* by Charles Daney

The REXX Reference Summary Handbook by Dick Goran

We offer a 60 day money back guarantee on all Quercus Systems' products.